ML/I User's Manual

Sixth Edition

R.D. Eager, P.J. Brown

This manual applies specifically to version CKL of ML/I. Some features may not be supported by earlier versions; conversely, later versions may support additional features.

Copyright © 1966,1967,1968,1970,1986,2004,2009,2010 R.D. Eager, P.J. Brown

Permission is granted to copy and/or modify this document for private use only. Machine readable versions must not be placed on public web sites or FTP sites, or otherwise made generally accessible in an electronic form. Instead, please provide a link to the original document on the official ML/I web site.

Table of Contents

1 Introduction 2

1.1 General description 2
1.2 Organisation of this manual 2
1.3 Notation for describing syntax 2
1.4 Further points of notation 3
1.5 Improving ML/I 3

2 The environment and its constituents 5

2.1 Basic action of ML/I 5
2.2 Character set 5
2.3 Text 5
2.4 Macros and delimiter structures 6
2.4.1 Examples of macros 7
2.4.2 Delimiter structures 7
2.4.3 Optional and repeated delimiters 8
2.4.4 Macro definitions 9
2.4.5 The difference between macros and subroutines . . . 9
2.4.6 Impossible replacements 9
2.5 Introduction to macro-time variables and statements 10
2.6 Inserts 10
2.6.1 Macro-time variables 11
2.6.2 Initialisation of macro variables 11
2.6.3 Subscripts and macro expressions 12
2.6.4 Character variables 13
2.6.5 Integer overflow 13
2.6.6 Macro labels 13
2.6.7 Macro elements 14
2.6.8 Insert definitions 14
2.6.9 Examples of inserts 15
2.7 Skips 16
2.7.1 Matched skips and straight skips 18
2.7.2 Literal brackets 18
2.7.3 Example of a matched skip 18
2.7.4 Warning markers 18
2.7.5 Stop markers 19
2.8 Summary of the environment 20
2.9 Normal-scan macros and straight-scan macros * 20
2.10 Name environment used for examples 21

3 Text scanning and evaluation 22
3.1 Nesting and recursion 22
3.2 Call by name 22
3.3 Details of the scanning process 22
3.4 The method of searching for delimiters 23
3.5 Exclusive delimiters * 24
3.6 Startlines * 25
3.7 Dynamically generated constructions * 26

4 Operation macros and their use 27
4.1 Operation macros 27
4.2 Use of literal brackets for surrounding operation macro arguments 27
4.3 NEC macros 28
4.4 Dynamic aspects of the environment * 29
4.5 Protected and unprotected inserts * 30
4.6 Ambiguous use of names * 30
4.7 Implications of rules for name clashes * 31

5 Specification of individual operation macros 33
5.1 Specification of delimiter structures 33
5.1.1 Keywords 34
5.1.2 The consequences of evaluation 35
5.1.3 Introduction to more complicated cases * 36
5.1.4 Full syntax of structure representations * 37
5.1.5 Examples of complex structure representations * 38
5.1.6 Possible errors in structure representations 39
5.2 The NEC macros 41
5.2.1 MCWARN 42
5.2.2 MCINS 43
5.2.3 MCSKIP 44
5.2.4 MCDEF 45
5.2.5 MCNOWARN, MCNOINS, MCNOSKIP and MCNODEF 47
5.2.6 MCWARNG, MCINSG, MCSKIPG and MCDEFG 48
5.2.7 MCSTOP 49
5.2.8 MCALTER 50
5.3 System functions 52
5.3.1 MCLENG 53
5.3.2 MCSUB 54
5.4 Further operation macros 55
5.4.1 MCSET 56
5.4.2 MCNOTE 57
5.4.3 MCGO 58
5.4.4 MCPVAR 61
5.4.5 MCCVAR 62

6 Error messages 63

6.1 Example of an error message 63
6.2 Notes on context print-outs 63
6.3 Count of errors 64
6.4 Complete list of messages 64
6.4.1 Illegal macro element 64
6.4.2 Arithmetic overflow 64
6.4.3 Illegal input character 65
6.4.4 Illegal macro name 65
6.4.5 Unmatched construction 65
6.4.6 Illegal syntax of argument value 66
6.4.7 Redefined label 66
6.4.8 Undefined label 67
6.4.9 Storage exhausted 67
6.4.10 System error 67
6.4.11 Subsidiary message 68
6.4.12 Statistics 68
6.4.13 Version number and current constructions 68
6.4.14 Implementation-defined messages 69

7 Hints on using ML/I 70

7.1 How to set up the environment 70
7.2 Possible sources of error 70
7.2.1 Jumping over expanded code 70
7.2.2 Generation of unique labels 70
7.2.3 Lower case letters 70
7.2.4 Use of newlines in definitions 70
7.2.5 Use of redundant spaces 71
7.3 Simple techniques 71
7.3.1 Interchanging two names 72
7.3.2 Removing optional debugging statements 72
7.3.3 Inserting extra debugging statements 73
7.3.4 Deleting a macro 73
7.3.5 Differentiating special-purpose registers and storage locations 73
7.3.6 Testing for macro calls 74
7.3.7 Searching 74
7.3.8 Bracketing within macro expressions 74
7.3.9 Deletion from source text only 75
7.3.10 Locating missing delimiters 75
7.3.11 Handling line-oriented input 75
7.4 Sophisticated techniques * 76
7.4.1 Macro-time loop 76
7.4.2 Examining optional delimiters 77
7.4.3 Dynamically constructed calls 78
7.4.4 Arithmetic expression macro 79
7.4.5 Formal parameter names 80
7.4.6 Intercepting changes of state 80

7.4.7 Remembering code for subsequent insertion 81
7.4.8 Constructions with restricted scopes 82
7.4.9 Optimising macro-generated code 83
7.4.10 Macro to create a macro 83

8 Use of system variables 85

8.1 System variable overview 85
8.2 Use of `S1` to `S9` 85

Operation Macro Index 88

Concept Index 89

Preface

This manual describes ML/I in full detail, with examples of its applications. It is not assumed that the reader has any previous knowledge of macro processors.

A shorter, simpler document describing ML/I is also available. This is called *The ML/I macro processor: a simple introductory guide*. There is also a tutorial document, called *An ML/I tutorial*, which is probably the best starting point for completely new users. A paper describing how ML/I is implemented appeared in *Communications of the ACM 13, 12 (December 1972), pp. 1059–1062*; the book *Macro Processors and portable software (Wiley, 1974)* contains further details.

Preface to the Sixth Edition

This edition is significantly changed from earlier editions. It incorporates, within the main text, all current and new features of ML/I. It has also been rewritten in Texinfo, so that it can be published in both printed and machine readable form; this has necessitated some re-wording and re-ordering of the text.

1 Introduction

1.1 General description

ML/I is a general macro processor. It is *general* in the sense that it can be used to process any kind of text. The text may be in any programming language or natural language, or it may be numerical data. The most important use of ML/I is to provide users with a simple means of adding extra statements (or other syntactic forms) to an existing programming language in order to make the language more suitable for their own field of application. This process of extension may be carried to the level where the extended language could be regarded as a new language in its own right. Other possible uses of ML/I are program parameterisation (e.g. a parameter might determine whether debugging statements are to be included in a program) and various applications in text editing or correction and in data format conversion.

This manual does not assume that the reader has any previous experience of macro processors. However, the reader who is familiar with macro processors might be interested in knowing the main features of ML/I before plunging into details. These features are:

- Macros with a variable number of arguments.
- Delimiters of the arguments of each macro are chosen by the user, and a macro may have several possible patterns of delimiters, each with a different meaning.
- Macro-time integer variables.
- Macro-time assignment and 'goto' statements.
- No restrictions on nesting and recursion.
- Macro calls occurring anywhere in the text (i.e. calls do not have to appear in a particular field, nor do they have to be preceded by a 'warning marker').
- Comprehensive error messages.

1.2 Organisation of this manual

Chapters 2, 3, 4 and 5 of this manual describe ML/I in full detail. Chapter 6 describes error messages, and Chapter 7 contains hints and examples. Readers may find it useful to look ahead to the examples in Chapter 7 if they have any difficulty with the main text. Some Sections of this manual can be omitted on a first reading; these are marked with an asterisk (like this: *). This manual does not describe features of ML/I that are implementation-dependent, e.g. operating instructions, character set, etc. Instead there is an Appendix, which describes the implementation-dependent features, for each implementation.

1.3 Notation for describing syntax

The notation used in this manual to describe syntax should be self-explanatory. An example of its use is the following description of a hypothetical `IF` statement:

```
IF {condition} THEN {statement};
```

As can be seen, a syntactic form is defined by concatenating its constituents. A constituent that is itself the name of a syntactic form is enclosed in braces (`{` and `}`). The remaining constituents are literals.

A well-known notation is used to indicate parts of syntactic forms that may optionally be repeated and/or omitted. In this notation, a constituent or series of constituents that may optionally be omitted is written:

```
[{constituents} ?]
```

Constituents that may be repeated any desired number of times are written:

```
[{constituents} *]
```

and constituents that may be omitted or repeated are written:

```
[{constituents} *?]
```

Thus, if the above `IF` statement had an optional `ELSE` clause, it would be written:

```
IF {condition} THEN {statement} [ELSE {statement} ?];
```

and a hypothetical `SUM` statement which permitted any number of arguments, provided there were at least two, might be defined:

```
SUM {argument} [,{argument} *];
```

Lastly, when there are several alternative forms for a constituent, these are written:

```
( {form 1} )
( {form 2} )
( { :: }   )
( { :: }   )
( {form N} )
```

Thus an expression might be defined as:

```
{variable} [ (+) {variable} *?]
             (-)
             (*)
             (/)
```

Note that the asterisk means that the syntactic forms enclosed within the brackets may be repeated; it is not required that identical text be written at each repetition.

1.4 Further points of notation

- When it is desired to emphasise the presence of a space, tab or newline in a piece of text, this is done by writing `{SPACE}`, `{TAB}`, or `{NL}` respectively. Note that this is simply a point of notation, and readers should be careful not to interpret an occurrence of, say, `{NL}` in a specification as requiring that they write the characters `{`, `N`, `L` and `}`.
- An integer is said to be positive only if it is greater than zero, and negative only if it is less than zero. Integers in ML/I are represented to a decimal base.

1.5 Improving ML/I

Readers are invited to criticise and suggest improvements in the specification of ML/I, in the description in this manual or in a particular implementation, and in particular to point out errors and ambiguities. Reports of implementation errors should be accompanied by enough material to reproduce the error and, if applicable, references to the statements in this manual that have been contravened.

2 The environment and its constituents

2.1 Basic action of ML/I

The basic action of ML/I is as follows. The user feeds to ML/I some text and an *environment*. The purpose of the environment is to specify that certain insertions, deletions, expansions, translations or other modifications are to be made in the text. ML/I performs the textual changes specified by the user. This process is called *evaluation of text*, and the text generated as a result of the changes is called the *value text*. The text being evaluated is called the *scanned text*. In many simple applications of ML/I, the process of evaluation consists of a good deal of straight copying, the value being the same as the original, but periodically a change is made and the generated value text is different from the original scanned text.

The purpose of this Chapter is to explain the mechanisms at the disposal of the user, and to give examples of their use. All the possible constituents of the environment will be described, and the resultant textual changes will be explained by describing the form of the scanned text and the form of the corresponding value in each case. The mechanisms for setting up the environment will be explained in subsequent Chapters.

2.2 Character set

The character set of ML/I (i.e. the set of allowable characters in the text it processes) is implementation-defined (see Section 3 of the relevant Appendix). However, the character set will normally contain the upper case letters A–Z, the lower case letters a–z, the numbers 0–9, and a number of characters that are not letters or numbers. Characters that are not letters or numbers are called *punctuation characters*. If an implementation contains both upper and lower case letters in its character set, then these are treated as entirely different sets of characters, and it is not possible to use a lower case letter interchangeably with its upper case equivalent. As will be seen, all the built-in symbols and characters that have special meaning to ML/I are in upper case (e.g. `MCDEF` rather than `mcdef`, and `S` rather than `s`).

2.3 Text

A feature of ML/I is that it does not consider text character by character, but in units of atoms. An *atom* is a single punctuation character, or a sequence of letters and digits that is surrounded by punctuation characters (assuming an imaginary punctuation character at the beginning and end of the text). There is no restriction on the length of an atom. To take an example, the text:

```
Pig , {TAB} LAC {SPACE} 4057
--- - ----- --- ------- ----

 1  2   3    4     5      6
```

would be regarded as six atoms as shown. It is possible to make an exception to this rule in special cases; see the description of system variable `S6` in Section 8.2 [Use of S1 to S9], page 85.

The following definitions will be used in the rest of this manual. *Text* is a (possibly null) sequence of atoms. The *source text* is the text supplied as input to ML/I, and the *output text* is the text derived from evaluating the source text. The physical form of the source text and output text is implementation-defined (see Section 2 of the relevant Appendix). The action of evaluating a particular piece of source text is called a *process*.

2.4 Macros and delimiter structures

Before defining a macro, it may be useful to consider the sort of text replacement that macros are designed to achieve. The assembly language for a hypothetical X123 machine will be used in several examples in this manual. Assume the user wants to introduce a new instruction of the form:

```
ESUB X      meaning 'subtract X from the accumulator'
```

which does not exist in the X123 instruction set, but whose effect can be achieved by the sequence of three X123 instructions:

```
CMA         complement accumulator
ADD X       add X to accumulator
CMA         complement accumulator
```

The introduction of `ESUB` would be achieved as follows. The user would write the program as if `ESUB` were an extra machine instruction. Before the program was assembled, it would be passed through ML/I with `ESUB` defined as a *macro name* with the above three instructions as its *replacement text*. ML/I would replace each occurrence of `ESUB` by its expanded form, and the resultant output could then be assembled normally. Each piece of text to be replaced is called a *macro call*, and the text corresponding to `X` above is called the *argument* of the call (within the replacement text of `ESUB` it is necessary to specify that the argument of the call should be inserted immediately after `ADD`. This is done by a constituent of the environment called an *insert*, which will be described later).

This example serves as a simple illustration of the primary use of ML/I, namely to serve as a preprocessor to an existing piece of software to allow users to introduce new statements of their own design into the existing language. Each new statement must be expandable in terms of the existing language.

Macros may have any number of arguments. Arguments are separated by predefined atoms (or sequences of atoms) called *delimiters*. When defining a macro, the user specifies what the delimiters are. The macro name is regarded as a delimiter, and is called the *name delimiter* to distinguish it from the remaining delimiters, which are called *secondary delimiters*. The delimiter following the last argument of a call is called the *closing delimiter*. The general form of a macro call can, therefore, be represented as:

```
{name delimiter} [{argument} {secondary delimiter} *?]
```

Arguments may be null, but delimiters must consist of at least one atom.

Every time ML/I encounters in the scanned text an atom or series of atoms that has been defined as a macro name, it searches for the secondary delimiters (if any) and then

replaces the entire macro call by the value of the replacement text for the macro. More details of the way macro calls are scanned are given in Section 3.4 [The method of searching for delimiters], page 23, and in Section 3.5 [Exclusive delimiters], page 24.

2.4.1 Examples of macros

It may be instructive at this stage to consider a few more examples of macros. These examples, which are listed below, are all of simple macros with fixed delimiters. Macros with more elaborate patterns of delimiters will be considered later. Note that ML/I could be used to add these macros to any desired programming language, whether high or low level.

Example 1

A macro to generate a loop, which has form:

```
DO {arg A} TIMES {arg B} REPEAT
```

Here the delimiters are `DO`, `TIMES` and `REPEAT`. `DO` is the name delimiter, and `TIMES` and `REPEAT` are secondary delimiters. `REPEAT` is the closing delimiter. ML/I does not require that macro calls be written on a single line, and calls of this macro would tend, in practice, to span several lines of text.

Example 2

A macro of form:

```
MOVE FROM {arg A} TO {arg B};
```

The name of this macro consists of the two atoms `MOVE` `FROM`.

Example 3

A macro to interchange two variables, which has form:

```
INTERCHANGE ( {arg A}, {arg B} ) {NL}
```

In this example, both the name and the closing delimiter consist of more than one atom: the name is `INTERCHANGE` followed by a left parenthesis, and the closing delimiter is a right parenthesis followed by a newline. Note that ML/I does not, like some software, truncate long names such as `INTERCHANGE`.

Example 4

Assume that within a program two different names, `COUNT` and `CONT`, have inadvertently been used for the same variable. This error could be corrected using ML/I, with `CONT` defined as a macro with `COUNT` as its replacement text. Here the name delimiter, `CONT`, is also the closing delimiter.

The reader should, at this stage, appreciate why ML/I considers text as a sequence of atoms rather than a sequence of individual characters. If the latter were the case, ML/I would be liable to take names such as `DOG` and `RANDOM` as calls of the above macro `DO`, since each name contains the letters `DO`. As the situation stands, however, the letters `DO` would only be taken as a macro call if they were surrounded by punctuation characters.

2.4.2 Delimiter structures

The macros considered so far have had fixed delimiters. However, it is possible to have macros with any number of alternative patterns of delimiters. As a very simple example of this, consider the `ESUB` macro. In X123 Assembly Language, statements are terminated with either a tab or a newline, and so it would be desirable to have both of these as alternatives for the closing delimiter of `ESUB`.

In order to specify the pattern of possible delimiters of a macro, the user specifies a *delimiter structure*. Each macro has its own delimiter structure, and other constituents of the environment also have delimiter structures. A delimiter structure is a set of delimiter specifications, each of which is a sequence of one or more atoms. These sequences of atoms need not be distinct. One or more of these delimiter specifications are designated as names of the structure. The remainder are secondary delimiters. With each delimiter specification is associated a specification of its *successor(s)*. This may be:

- null,
- another delimiter specification within the structure,
- a set of alternative delimiter specifications within the structure.

Successors specify what to search for next when scanning. A delimiter with a null successor is a closing delimiter. As an illustration of the use of a delimiter structure, consider the scanning of a macro call. During this scanning, each time a delimiter is found, the delimiter structure of the macro being called is referenced to find the successor(s) of the current delimiter, and subsequent text is then scanned to try to find this successor. This process continues until a closing delimiter is found.

As an example of a delimiter structure, the delimiter structure of the `ESUB` macro would contain three delimiter specifications with the following information about them:

a)	`ESUB {`*`name`*`}`	with b) or c) as its successor.
b)	`{`*`TAB`*`}`	secondary delimiter with no successor.
c)	`{`*`NL`*`}`	secondary delimiter with no successor.

The rules for setting up delimiter structures (see Section 5.1 [Specification of delimiter structures], page 33) ensure that they have certain properties. Among these properties are the following:

- If there is more than one name, each name is represented by a different sequence of atoms.
- If a delimiter structure has alternative successors, each is represented by a different sequence of atoms.
- The structure is connected. This means that it must be possible to reach each secondary delimiter by a sequence of successors from some name.

2.4.3 Optional and repeated delimiters

It is possible, by designing a suitable delimiter structure, to have a macro with a variable number of arguments; in particular, a macro with optional arguments and/or with an indefinitely long list of arguments. For instance, suppose it is desired to implement a macro with alternative forms:

```
IF {argument} THEN {argument}
END
```

and

```
IF {argument} THEN {argument}
ELSE {argument}
END
```

This is done by specifying that either `ELSE` or `END` is the successor of `THEN`. `END` is a closing delimiter, and `ELSE` has successor `END`. As a second example, consider a macro of form:

```
SUM {argument} [ (+) {argument} *?];
                 (-)
```

This macro has an indefinite number of arguments, separated by plus or minus signs. Its delimiter structure has four members as follows:

a)	`SUM`	name with b), c) or d) as successor.
b)	`+`	secondary delimiter with b), c) or d) as its successor.
c)	`-`	secondary delimiter with b), c) or d) as its successor.
d)	`;`	secondary delimiter with no successor.

2.4.4 Macro definitions

Now that the basic concepts behind macros have been introduced, it is possible to explain more exactly what makes up a *macro definition*. Macro definitions are the most important constituents of the environment. A macro definition consists of:

a. A delimiter structure. The name delimiter(s) of this structure are the macro names.

b. A piece of replacement text.

c. An integer, exceeding two, called the *capacity*. The purpose of this is explained in Section 2.6.1 [Macro-time variables], page 11.

d. An on/off option. If this option is on, the macro is called a *normal-scan* macro; otherwise it is called a *straight-scan* macro. The effect of this option is explained in Section 2.10 [Name environment used for examples], page 21.

The reader need not for the moment be concerned with (c) and (d), since nearly all macros will be normal-scan and will have a capacity of three.

2.4.5 The difference between macros and subroutines

There is often confusion between the purpose of macros and the purpose of subroutines (or procedures). Macros, however, always generate in-line code, and so this code is inserted as many times as the macro is called. Subroutines use out-of-line code, and there is only one copy of this code for a particular program. Thus, macros are used only when the code to be inserted is short or highly parameterised. It would not be convenient, for instance, to use subroutines to perform the functions of any of the macros used as examples in previous Sections.

2.4.6 Impossible replacements

It is worth noting some of the types of replacement that it is not possible to perform by means of macros. Below are two examples of illegal syntax of macro calls, together with possible correct forms.

a. *Wrong*: `{arg A} = {arg B};`
since each macro call must start with a macro name.

Right: `SET {arg A} = {arg B};`
Here SET is used as the macro name.

b. *Wrong*: `$ {character}`
It is not possible to define an argument as the character (or atom) immediately following a given name. Every argument must be followed by some predefined delimiter.

Right: `$ {argument};`
Here a semicolon is used as the closing delimiter.

2.5 Introduction to macro-time variables and statements

The form of the value of a call of such macros as the `IF` and `SUM` macros used earlier as examples would have to depend on the particular patterns of delimiters that were used in the call. For instance:

```
SUM ALPHA+BETA;
```

must generate an entirely different set of instructions from:

```
SUM ALPHA-BETA-GAMMA+X+Y-Z;
```

and, in the case of `IF`, the value text must depend upon whether `ELSE` was present. Macros such as these, therefore, are more complicated than the `ESUB` case, where a fixed skeleton of code consisting of three machine instructions is substituted for each call. The only variable element in the `ESUB` case is the form of its argument. In the more complicated cases, where the delimiters provide a second variable element, the user has to write a little program which is executed by ML/I and tests the form of the delimiters used and generates code accordingly. In the case of `SUM`, which has an indefinitely long list of arguments and delimiters, this program would involve a simple repetitive loop to iterate through the list. Hence, ML/I contains an elementary programming language of its own. This language contains an assignment statement, a conditional 'goto' statement, labels, and integer and character string variables. All of these are called *macro-time* entities to distinguish them from the corresponding *execution-time* entities, and the reader must be careful not to confuse the two. The difference is illustrated thus: the `DO` macro described earlier (see Section 2.4.1 [Examples of macros], page 7) would generate a loop which was performed at execution time and controlled by an execution-time variable; on the other hand the value text for the `SUM` macro would be *generated by* a macro-time loop controlled by a macro-time variable.

Macro variables and macro labels are considered in the next Section. Macro-time statements are considered in detail in Chapter 4 [Operation macros and their use], page 27.

2.6 Inserts

This Section describes how quantities can be inserted into text. In particular, it describes how arguments of macro calls are inserted into replacement text. However, first it is necessary to consider some of the quantities, in addition to arguments, that may be inserted into text.

2.6.1 Macro-time variables

Macro variables are integer or character variables available to the user at macro-time. ML/I contains facilities for performing arithmetic on these variables (where appropriate), testing their values, and inserting their values into the text. They are useful as switches and for counting (e.g. in processing macros with a variable number of arguments), as well as for storing information which may be needed later on in a process.

There are four kinds of macro variable, namely:

a. *permanent variables*, referred to as `P1`, `P2`, ...
b. *system variables*, referred to as `S1`, `S2`, ...
c. *temporary variables*, referred to as `T1`, `T2`, ...
d. *character variables*, referred to as `C1`, `C2`, ...

Permanent, temporary and system variables are used to store integers. Character variables are used to store character strings.

Permanent, system and character variables have *global scope*; this means they can be referred to anywhere. An implementation-defined number of each is allocated at the start of each process, and these remain in existence throughout. The user may allocate extra permanent variables and character variables (but not system variables) if desired, see Section 5.4.4 [MCPVAR], page 61, and Section 5.4.5 [MCCVAR], page 62. The difference between permanent/character and system variables is that the former have no fixed meanings and are free for users to use as they wish, but the latter have fixed implementation-defined meanings associated with controlling the operation of ML/I. For example, in a given implementation, system variable `S20` might control the listing of the source text; if it was zero no listing would be produced and if it was one there would be a listing. Sections 5 and 7 of each Appendix describe the meanings of system variables (if any) and state the number of permanent and system variables that are initially allocated. System variables 1 to 9 are normally the same in most implementations, and are described in Chapter 8 [Use of system variables], page 85.

Temporary variables, on the other hand, have a more local scope. During the evaluation of the source text there are no temporary variables in existence. However, each time a macro call is made a number of temporary variables is allocated, and these remain in existence while the replacement text of the macro is being evaluated. The number of temporary variables allocated at the call of a macro is given by the capacity of the macro (see Section 2.4.4 [Macro definitions], page 9). The capacity is usually three. If temporary variable N is referenced during the evaluation of the replacement text of a macro call, this is taken to mean the Nth temporary variable associated with the call. Since, as will be seen later, it is possible to have macro calls within macro calls, it is possible to have several allocations of temporary variables in existence at the same time.

2.6.2 Initialisation of macro variables

The initial values of system variables, permanent variables and character variables are defined in Section 7 of each Appendix. The first three temporary variables of each allocation are initialised as follows (all other temporary variables have undefined initial values):

`T1` the number of arguments of the current macro call.

`T2` the number of macro calls so far performed by ML/I during the current process. The importance of this number is that it is unique to the current call.

`T3` the current depth of nesting of macro calls (i.e. the number of calls, including the present one, currently being processed; calls of operation macros (see Section 4.1 [Operation macros], page 27) are not counted here, though they do count toward the setting of `T2`).

It is to be emphasised that these are initial values, and the user is free to change them if desired (in this way, temporary variables are unlike system variables. If the values of system variables—even those without assigned meanings—were changed arbitrarily it might have unwanted effects).

2.6.3 Subscripts and macro expressions

In the previous Sections, macro variables were specified by a letter followed by a number (e.g. `P2`), but there are other possibilities. The general form of a macro variable is:

```
(P)
(S) {subscript}
(T)
(C)
```

where a *subscript* is an unsigned positive integer, or an integer macro variable (but not a character variable). The value of the subscript specifies the macro variable to be referenced. Thus, if `T3` has value 4, then `PT3` would specify `P4`. As a more complicated example, if `T1` had value 2 and `P2` had value 6, then `TPT1` would specify the sixth temporary variable. If character variable 3 contained the string `16`, it would be wrong to write `PC3` in order to reference permanent variable 16; however, it would be possible to obtain the same effect by using an insert, in which case the user would write `P%C3.` instead.

Macro variables can be combined into *macro expressions*, which are used when it is desired to perform arithmetic calculations during macro generation. Examples of macro expressions are:

```
1, -6, 3-S1, -TT1-145/P2+P3+6
```

Multiplication is represented by an asterisk, and division by a slash. Bitwise logical operations are also supported; ampersand (`&`) is used for logical "and", and vertical bar (`|`) for logical "or". The general form of a macro expression is:

```
{primary} [ (+) {primary} *?]
          (-)
          (*)
          (/)
          (&)
          (|)
```

where a *primary* has the form:

```
[ (+) *?] {operand}
```

and an *operand* is an unsigned integer or an integer macro variable. Redundant spaces can occur anywhere in macro expressions except within operands.

The *result* of a macro expression is the integer derived from calculating the expression by the ordinary rules of arithmetic. Unary operators are performed first, followed by the binary operators from left to right, with the proviso that multiplication and division take precedence over addition and subtraction. Division is truncated to the greatest integer that does not exceed the exact result. Division by zero is detected as an error. Examples of the results of macro expressions are:

`1 + 2 * 3`	has result 7
`3 * 7/8`	has result 2
`7/8 * 3`	has result 0
`- 5/4`	has result −2
`5/-4`	also has result −2
`- 4/-3 * -6`	has result −6

2.6.4 Character variables

Strictly speaking, character variables should be called *character string variables*, or just *string variables*, but since their names begin with the flag character `C`, rather than `S` (which is already taken for system variables), we shall persist with the original term.

Character variables can store character strings of any length, up to a maximum known as the *range*. For a given ML/I process, all character variables have the same range, chosen by the user when initially allocating the first character variable. When the value of a character variable is inserted, the length of the inserted text will be the length of the string last stored in the variable, rather than the range (i.e. no trailing spaces are included unless originally stored as such).

Character variables are particularly useful as an efficient solution to the problem of 'remembering' pieces of text which are to be recalled at some later point in the ML/I process. An alternative solution would be to use the `MCFOR` macro described in Section 7.4.1 [Macro-time loop], page 76.

2.6.5 Integer overflow

Each implementation has a maximum absolute value which must not be exceeded by any integer derived during the calculation of a macro expression or subscript. The effect of exceeding this value is implementation-defined. See Section 5 of the relevant Appendix for details.

2.6.6 Macro labels

Since there is a facility for a macro-time 'goto', there is also a facility for placing macro-time labels. These are called *macro labels*. Each macro label is designated by a unique positive integer.

2.6.7 Macro elements

Macro variables, macro labels, arguments and delimiters are collectively called *macro elements*. It is convenient to regard macro elements as part of the environment. The full details of how macro elements are added to the environment are explained later; see Section 4.4 [Dynamic aspects of the environment], page 29. In essence, the rule is that every time a macro is called its arguments and delimiters, plus a set of temporary variables, are automatically added to the environment, and this supplemented environment is used to evaluate the replacement text of the call. Similarly, when a macro label is encountered, its position is 'remembered' by adding it to the environment.

2.6.8 Insert definitions

It is now possible to define the constituent of the environment, called an *insert definition*, which is used for such purposes as to tell ML/I to insert a particular argument of a macro at some point in its replacement text. An insert definition consists of:

a. A delimiter structure. Since all inserts have fixed delimiters and exactly one argument, this delimiter structure will be a simple one. It will consist of a name with a single successor, this successor being a closing delimiter.

b. An on/off option. If this option is on, an insert is called *protected*; otherwise it is called *unprotected*. The use of this option, which need not be of much concern to the average reader, is described later; see Section 4.5 [Protected and unprotected inserts], page 30.

At each point where the user wishes something to be inserted, they should write the following construction, called an *insert*:

```
{insert name} {argument} {delimiter}
```

In the rest of this manual, for the purpose of examples, it will be assumed that the atom `%` is an insert name, with the atom `.` as its closing delimiter. With this assumption, the following are examples of inserts (the exact meaning of these will become apparent later):

```
%A6.  %P1.  %LT2.  %WA P9-16*T3.
```

On encountering an insert, ML/I evaluates the argument of the insert (in case it contains macro calls, etc.) and the resulting value text acts as a specification of what to insert. The value text must consist of a *flag* followed by a macro expression. In the first above example, the flag is `A` and the macro expression is `6`. The flag may be null, or it may be any of the following: `A`, `B`, `D`, `L`, `WA`, `WB` or `WD`. Any number of redundant spaces is allowed before, after or within a flag.

The meaning of the various flags is explained below. In each explanation, 'N' is used to represent the value of the macro expression following the flag. More examples are given in the next Section. An attempt to insert something which does not exist (e.g. the third

argument of a macro with only two arguments) results in an error. The meanings of the flags are:

a. `A`. This flag is used within the replacement text of a macro to evaluate and insert the Nth argument of a call of the macro. Any spaces at the beginning or end of the argument are deleted before it is evaluated. In the case of this flag, and in cases b) and c) below, the piece of text that is evaluated and inserted is called the *inserted text*.

b. `B`. As case a), except that spaces are not deleted.

c. `D`. As case b), except that the Nth delimiter, rather than the Nth argument, is inserted. The name of a macro is considered as delimiter zero, and the Nth delimiter is thus the delimiter following the Nth argument.

d. `WA`, `WB`, `WD`. As cases a), b) and c) respectively, except that the inserted text is not evaluated but is inserted literally, exactly as written (`W` stands for 'written'). The difference between this and the previous cases arises if the inserted text itself involves macro calls, inserts, etc. In the previous cases these are evaluated; in this case they are not.

e. Null. The numerical value of N, represented as a character string, is inserted. This character string contains no redundant leading zeros. It is preceded by a minus sign if N is negative; otherwise no sign is present.

f. `L`. This is used to place a macro label, and is rather different from the above cases in that nothing is inserted (i.e. the value of the insert is null). The label N is, if acceptable, added to the current environment and may be the subject of a macro-time 'goto'. A macro label is acceptable if it is inserted within a piece of replacement text or inserted text and has not already been defined within that text. It is legal to insert a label in the source text, but since, as will be seen later, it is not possible to have a backward 'goto' within the source text, such labels are not added to the environment (i.e. they are 'forgotten'). Macro labels are local to the piece of text in which they occur, and there is no harm in using the same label numbers within different pieces of text. Label numbers can be chosen arbitrarily, except that they must be positive.

2.6.9 Examples of inserts

The following examples illustrate the use of inserts:

- The replacement text of the `ESUB` macro (see Section 2.4 [Macros and delimiter structures], page 6) might be written:

```
CMA
ADD     %A1.
CMA     {NL}
```

  or even:

```
CMA
ADD     %A1.
CMA     %D1.
```

  The latter form would have the advantage of inserting newline or tab, according to which one was written in the call.

- In the case of the `DO` macro (see Section 2.4.1 [Examples of macros], page 7), the replacement text would involve an execution-time label. It is imperative that a different

execution-time label be generated for each call of DO. This could be achieved by using the initial value of T2. The label could, for example, be written:

```
ZZ%T2.
```

In this case, if two successive calls of DO occurred at the start of the source text, then ZZ1 would be generated at the first call and ZZ2 at the second.

- If SWITCH is a macro name with replacement text P1, then it is possible to write:

  ```
  %SWITCH.
  ```

  to insert the first permanent variable. The reason is that the argument of an insert is evaluated before being processed, and the call of the SWITCH macro would be performed during this evaluation.
- The occurrence of %A1. in the replacement text of the macro call:

  ```
  MOVE FROM JACK TO JOHN;
  ```

  would cause JACK to be inserted, whereas the occurrence of %B1. would cause JACK enclosed in spaces to be inserted.
- If it is desired to insert the name of a macro into its replacement text, this can be done by writing %WD0. (the reason for this facility is that macros can have several alternative names). In general it would be wrong to use %D0. instead, since this form causes any macro calls within the delimiter to be performed. But delimiter zero is the macro name itself, and hence an endless recursive loop is likely. In fact, when inserting delimiters it is usually better to use a W.
- This example rather jumps the gun in that it uses the macro-time statements MCSET and MCGO which have not yet been defined. However, if the reader cares to try to understand this example at this stage, it may give a useful insight into the purpose of the preceding material. The example shows how the replacement text of the SUM macro could be written—the comments at the side are for the reader's benefit and do not form part of the replacement text. The blank lines are not part of the replacement text, either.

```
LAC %A1.                        Generate code to load accumulator
                                with first argument.

MCSET T2 = 1                    Use T2 as loop counter.
%L4.MCGO L1 IF %DT2. = +        Test if current delimiter is plus ...
MCGO L2 IF %DT2. = -            ... or minus.
MCGO L0                         If neither then exit (L0 has a
                                special meaning, namely 'return').
%L2. ESUB %AT2+1.               Generate code to subtract the
                                current argument.

MCGO L3
%L1. ADD %AT2+1.                Generate code to add current argument.
%L3.MCSET T2 = T2 + 1           Increase T2 and continue loop.
MCGO L4
```

2.7 Skips

The description so far has implied that every occurrence of a macro name in the scanned text is taken as the start of a macro call. This would mean that the user had no easy means

of getting macro names or, for that matter, insert names into the value text. Moreover, if he or she were unfortunate enough to use a macro name within any comments, then ML/I would take this as a macro call and would start searching for delimiters. To get round these difficulties the user places *skip definitions* in the environment, and by this means can cause ML/I to ignore comments and to take certain strings as literals.

A skip definition consists of:

a. A delimiter structure. The names of this structure are called *skip names*.

b. Three on/off options. These options are: the *text option*, the *delimiter option* and the *matched option*.

The action of ML/I on finding a skip name is similar to the action on finding a macro name. In both cases a search for delimiters is made until a closing delimiter is found. The text from the skip name to its closing delimiter is called a *skip*. A skip, therefore, has form:

```
{skip name} [{argument} {secondary delimiter} *?]
```

In most practical applications of skips, there will be exactly one argument. The arguments of skips are treated as literals, exactly as if all macro definitions, insert definitions and warning markers (see later) had been temporarily removed from the environment during the scanning of the skip. There is no replacement text associated with a skip; instead the value of a skip is defined simply by the setting of two of its options. These options, which are independent of one another, have the following effect:

- If the *delimiter option* is on, then the delimiters of the skip are copied over to the value text; otherwise they are not.
- If the *text option* is on, then the arguments of the skip are copied over to the value text; otherwise they are not.

As an example of the use of a skip, assume the source text contains comments that begin with the word `COMMENT` and end with a semicolon. In order to skip these comments, the user would define `COMMENT` as a skip name with semicolon as its closing delimiter. In this case, if the following comment occurred:

```
COMMENT THIS DO LOOP ZEROISES ARRAY X;
```

then its value (i.e. the piece of text copied over to the value text) would be one of the following:

a. If both options were on, its value would be:

```
COMMENT THIS DO LOOP ZEROISES ARRAY X;
```

b. If neither option was on, its value would be null.

c. If only the delimiter option was on, its value would be:

```
COMMENT;
```

d. If only the text option was on, its value would be:

```
THIS DO LOOP ZEROISES ARRAY X
```

If `COMMENT` was not defined as a skip at all, then comments would normally be copied over to the value text as in case a). However, if in the above example `DO` was a macro name, then ML/I would try to find the delimiters of `DO` and replace the call of `DO` by its replacement text. This is clearly undesirable. The chances are that the entire source text would be scanned

without finding the required delimiters. Hence the use of skips to inhibit the recognition of macro names within certain contexts.

It will be assumed in the rest of this manual that `COMMENT` is a skip name, with a semicolon as its closing delimiter.

2.7.1 Matched skips and straight skips

Assume the user has written the comment:

```
COMMENT THIS COMMENT MARKS THE HALF-WAY STAGE;
```

In this case, the skip name `COMMENT` appears within an argument of the skip `COMMENT`. However, it is clearly undesirable that ML/I should treat the second `COMMENT` as a nested skip and try to match it with a semicolon. To prevent this happening, `COMMENT` would be defined as a skip with the matched option off. This is called a *straight skip*.

However, there are applications of skips where it is desirable for nested skips to be recognised, and such skips have the matched option on. They are called *matched skips*. *Literal brackets*, which are described in Section 2.7.2 [Literal brackets], page 18, are an example of the application of a matched skip. If ML/I encounters any skip name during the scanning of a matched skip, it matches the nested skip with its delimiters before matching the containing skip with its delimiters. The scanning process is described in more detail in Section 3.4 [The method of searching for delimiters], page 23. In a nest of skips, the value is entirely controlled by the options associated with the outermost skip.

2.7.2 Literal brackets

It is usual to have in each environment a skip definition consisting of a name and a closing delimiter with the options set in such a way that at every occurrence of the skip the argument is copied and the delimiters deleted. Such skips are called *literal brackets*. It will be assumed in the rest of this manual that the name `<` with closing delimiter `>` have been defined as a pair of literal brackets. If it was required to copy a piece of text literally over to the value text, ignoring all macro calls and inserts, then the text would be written:

```
< {text} >
```

The process of evaluation would consist simply of removing the literal brackets. Literal brackets always have the matched option on. The reason for this will become apparent in Section 4.2 [Use of literal brackets for surrounding operation macro arguments], page 27.

2.7.3 Example of a matched skip

The following example, which is rather more complicated than any situation likely to arise in practice, illustrates the full implications of the rules for the matching of skips.

In the text:

```
< AAA < BBB COMMENT < ; CCC > DDD >
```

the initial `<` is matched with the last `>`. (The occurrence of `<` after `COMMENT` is not recognised as a skip name, since `COMMENT` is a straight skip). The value of this text is:

```
AAA < BBB COMMENT < ; CCC > DDD
```

This value is independent of how the delimiter and text options for `COMMENT` are set.

2.7.4 Warning markers

Up to now, ML/I has been described as if every occurrence of a macro name not within a skip is taken as the start of a macro call. In fact, this is only true if the environment is in *free mode*.

If desired, the user may place the environment in *warning mode* by defining one or more *warning markers*. Any atom or series of atoms may be defined as a warning marker. In warning mode, each macro call must commence with a warning marker. Optional spaces are allowed between the warning marker and the macro name which follows it; this means that in warning mode it is not possible to have a macro name that begins with a space. Thus, if `CALL` were a warning marker, the `ESUB` macro would be called by writing:

```
CALL ESUB X {NL}
```

In warning mode, each occurrence of a warning marker must be followed by a macro name; failure to do so will result in an error message. The message can be suppressed by setting system variable `S3` (see Section 8.2 [Use of S1 to S9], page 85) to one. This is useful if macro calls in the source text are only to be recognised in certain positions, e.g. following a tab or at the start of a line. In such examples the characters 'tab' or 'startline' could be defined as warning markers, and, assuming that not all occurrences need to be followed by macro calls, `S3` could be set to one to suppress the message.

Note that, if a warning marker is not followed by a macro name, it is treated as if it were not a construction name at all and is thus normally copied over to the value text. This applies irrespective of whether `S3` is being used to suppress the error message.

The essential difference between warning mode and free mode is that in the first case all macro calls have to be specially marked by preceding them with warning markers, whereas in the second case all macro names that are not to be taken as macro calls have to be specially marked by enclosing them in skips.

Note that warning markers only apply to macro calls, and must not be used to precede inserts or skips. These latter are always recognised, irrespective of the mode of the scan.

2.7.5 Stop markers

Normally, if a delimiter of a macro call in the source text is accidentally omitted or wrongly specified, then the remainder of the source text might be scanned over in searching for the missing delimiter.

Thus, there is a construction called a *stop marker*. Stop markers are only recognised when searching for a delimiter of a construction in the source text. Outside of this context, stop markers are not part of the environment. If it encounters a stop marker, ML/I gives a message to signal that the current construction(s) are *unmatched*. The text from the construction name up to (but not including) the stop marker is ignored, and scanning is resumed at the stop marker itself. For example, if the source text read:

```
MCDEF IF THEN NL
AS <. . .>
```

and newline were then declared as a stop marker, and further source text included:

```
IF X = Y THIN GO TO Z
```

then ML/I would take the final newline as a stop marker and would give the error message:

```
Delimiter THEN of macro IF in line . . . not found
```

Stop markers obey the normal rules for name clashes, See Section 4.6 [Ambiguous use of names], page 31. Hence if, in the above example, THIN were replaced by THEN, then the final newline would be treated as a delimiter of IF rather than as a stop marker, and there would be no error message. An implication of this is that if the following definition were added to the above text:

```
MCSKIP DT, COMMENT N1 OPT NL N1 OR ; ALL
```

then

```
COMMENT XXX
YYY
ZZZ;
```

would not cause an error since all newlines would be treated as delimiters, not stop markers. In general, therefore, it is possible (though tortuous in all but the simplest cases) to define constructions that may be arbitrarily long even if stop markers have been defined.

Note that stop markers override the normal scope rules in that they are recognised within skips and within straight-scan macros. They are treated as local constructions.

Stop markers will stop forward searches for labels in the source text, as well as the scan for unmatched constructions.

2.8 Summary of the environment

All the constituents of the environment have now been defined. To recap, these are:

a. Macro definitions.
b. Insert definitions.
c. Skip definitions.
d. Warning marker definitions.
e. Stop marker definitions.
f. Permanent variables.
g. Character variables.
h. System variables.
i. Temporary variables.
j. Arguments.
k. Delimiters.
l. Macro labels.

The term *construction* is used as a collective name for skips, inserts and macro names, and the term *name environment* is used as a collective name for constituents a), b), c), d) and e) above, since the names of these constituents are used to recognise constructions in the scanned text.

2.9 Normal-scan macros and straight-scan macros *

This Section explains the difference between normal-scan macros and straight-scan macros. However, straight-scan macros have only limited uses and the reader may choose to skip this Section and assume that all macros are normal-scan.

The difference between the two types of macro arises in the scanning of macro calls. In the case of a normal-scan macro, constructions nested within the call are recognised; in the case of a straight-scan macro, the effect is as if the name environment were temporarily removed during the scanning of the call. As an example of the use of a straight-scan macro, consider a language where comments are commenced with the word `NOTE` and ended with a semicolon. Assume it is desired to use ML/I to map this language into a language where comments are enclosed between the atoms `[` and `]`. It is not possible to achieve this transformation by the use of skips, since the options on skips do not permit the insertion of extra characters; moreover normal-scan macros are inadequate since it is not desired to recognise macro names within comments. Hence `NOTE` would be defined as a straight-scan macro. Its replacement text would be:

```
[%WA1.]
```

The replacement text of a straight-scan macro is evaluated in exactly the same way as that of a normal-scan macro.

The reader will no doubt have noticed that there is an analogy between the two types of macro and the two types of skip. In fact, any straight skip can be represented as a straight-scan macro. However, straight skips are preferable, where possible, since they are slightly easier to define and much faster in execution. The analogy between normal-scan macros and matched skips is not so close. Normal-scan macros permit any constructions to be nested within calls of them, whereas matched skips only allow further skips to be nested within them.

The straight-scan option can only apply to user-defined macros; it cannot apply to inserts or to operation macros (see Section 4.1 [Operation macros], page 27).

2.10 Name environment used for examples

To avoid unnecessary repetition, a fixed name environment will be assumed in all subsequent examples. This environment consists of:

- The atom `%` with closing delimiter `.` as an insert definition.
- The atoms `<` and `>` as literal brackets.
- `COMMENT` as a straight skip with closing delimiter semicolon.
- The `DO` and `MOVE` macros of Section 2.4.1 [Examples of macros], page 7.
- The `ESUB` macro of Section 2.4 [Macros and delimiter structures], page 6.
- No warning markers.
- No stop markers.

All the macros above are taken to be normal-scan.

3 Text scanning and evaluation

3.1 Nesting and recursion

Constructions may be nested to any desired depth, and may appear within replacement text. Furthermore, recursive macro calls are allowed. In other words, any construction is allowed with any piece of replacement text or inserted text, and a macro may be called while evaluating its own replacement text. However, constructions must be properly nested. This means that each construction must lie entirely within a single piece of replacement text, entirely within a single piece of inserted text or entirely within the source text. Apart from this obvious restriction, ML/I contains no restrictions on nesting and recursion.

As a result of nesting and recursion, the process of text evaluation is normally a recursive one. At the beginning of a process, ML/I starts evaluating the source text. During this evaluation, it will in general encounter a macro call. This will cause it temporarily to suspend the evaluation of the source text and start evaluating the replacement text of the call. While evaluating this replacement text, ML/I may encounter an insert, and this will cause it to suspend the evaluation of the replacement text and start evaluating some inserted text. Alternatively, it may encounter a nested macro call. Thus at any one time several pieces of text may be in the process of evaluation.

This situation is liable to lead to ambiguities in terminology, so it is necessary to clarify some of the terms that will be used. The terms *scanned text*, *current environment* and *current point of scan* will always refer to the text actually being evaluated, not to any piece of text whose evaluation has been temporarily suspended. ML/I is said to be *evaluating inserted text* if the scanned text is inserted text, and a similar definition applies to *evaluating replacement text*. ML/I is said to be *evaluating the source text* if it is not within the evaluation of any macro calls or inserts.

3.2 Call by name

Arguments and delimiters are evaluated each time they are inserted, rather than when the call in which they occur is scanned. In other words, they are 'called by name' rather than 'called by value'. In most cases, of course, this choice of approach makes no difference to the final result, but it does have an effect if the environment changes between the time an argument is scanned and the time it is inserted.

3.3 Details of the scanning process

When text is evaluated, it is scanned atom by atom until the end is reached. All text, whether the source text, replacement text or inserted text, is scanned and evaluated in the same way. In general, each atom of the scanned text is compared with all the names in the environment to see if a match can be found. However, as was seen in the previous Chapter, some types of name are not recognised under certain circumstances. The complete list of such circumstances is as follows:

a. No names are recognised within a straight skip or a straight-scan macro call.

b. Apart from skip names, no names are recognised within a matched skip.

c. In warning mode, macro names are not recognised except after warning markers. Immediately after a warning marker, no names except macro names and no secondary delimiters are recognised unless an error occurs (see Section 6.4.4 [Illegal macro name], page 65).

When a construction name is found, a search is made for its closing delimiter. This process is described in Section 3.4 [The method of searching for delimiters], page 23.

Some names in the environment may consist of more than one atom. In this case, when an atom of the scanned text is found to match the first atom of the name, the scanning process looks ahead to see if the remaining atoms of the name follow this atom (this look-ahead is abandoned if the end of the current text is reached). If a match is found, scanning is resumed beyond the last atom of the name. The user can specify, for each pair of atoms of a multi-atom name, whether spaces between the atoms are to be ignored by the scan. Multi-atom secondary delimiters are matched in exactly the same way as multi-atom names.

Apart from these cases of multi-atom delimiters, the scan always proceeds atom by atom. Each atom not within a construction is copied over to the value text. Atoms within skips may or may not be copied according to the option settings. Atoms within macro calls or inserts are never copied over to the value text since the very purpose of these constructions is to perform a replacement.

3.4 The method of searching for delimiters

When ML/I encounters a construction name, it searches for each of the secondary delimiters until the closing delimiter is found (except in the case where the construction name is its own closing delimiter, when no searching is required). In general, an error message (see Section 6.4.5 [Unmatched construction], page 65) is given if the end of the current piece of text is reached before the closing delimiter has been found. In this case the construction is said to be *unmatched*. *Exclusive delimiters*, however, provide a slight exception to this rule (see Section 3.5 [Exclusive delimiters], page 24). If, during the search for the delimiters of a construction, a nested construction is encountered, then the search for the delimiters of the outer construction is suspended until the closing delimiter of the nested construction has been found. Nested constructions can only arise within inserts, matched skips and normal-scan macros. Since arguments are called by name rather than by value, nested constructions are not evaluated when scanned over during the search for delimiters of a containing construction. Evaluation occurs only when the argument containing the nested construction is inserted.

The process of searching for closing delimiters is illustrated by the following rather pathological example (remember that the name environment of Section 2.10 [Name environment used for examples], page 21, applies to this and all subsequent examples).

```
DO 3 TIMES < REPEAT DO >
     ESUB REPEAT
     DO REPEAT TIMES
     REPEAT
REPEAT
```

In this example the first DO is matched with the last REPEAT, since the search for the REPEAT for this first DO is suspended during the scanning of the nested constructions <, ESUB and DO. Furthermore, the occurrence of DO within the literal brackets is not recognised as a macro name.

In general, a single closing delimiter cannot terminate two separate constructions. Thus, two successive REPEATs are needed in the above example to close both the DO macros. However, exclusive delimiters again provide an exception to the rule.

As a further example, if the user were foolish enough to write:

```
MOVE FROM TO TO PIG;
```

then the first TO would be taken as the delimiter of MOVE FROM. What should be written to make the second TO the delimiter is:

```
MOVE FROM <TO> TO PIG;
```

However, there is nothing wrong with writing:

```
MOVE FROM PIG TO TO;
```

In practice, if delimiter names are chosen sensibly, problems such as the above rarely arise.

3.5 Exclusive delimiters *

It is highly recommended that this Section be skipped on a first reading, as it describes a rather complicated feature which is only occasionally needed.

In the normal way, after a construction has been scanned over and replaced by its value, scanning is resumed with the atom following the closing delimiter of the construction. Hence the closing delimiter is taken as part of the construction. In a few cases, however, it is more convenient to regard the closing delimiter as external to the construction. Such a delimiter is called an *exclusive delimiter*. Only macros and skips may have exclusive delimiters, and exclusive delimiters are always closing delimiters. After a construction with an exclusive delimiter has been dealt with, scanning is resumed *at* the exclusive delimiter rather than beyond it.

Exclusive delimiters are useful when it is desired to use a single delimiter as a closing delimiter of several nested constructions. For example, an IF macro might have form:

```
IF {condition} THEN {nested macro call} {NL}
```

where the nested macro call is terminated, like IF, by the closing newline. In this case, it would be necessary to define newline as an exclusive delimiter of any macro that could be nested within the IF macro. Then, when the scan had used the newline to close the nested macro call, it would re-scan it and use it again to close the IF macro.

A difficulty arises in the above example when, within the replacement text of IF, the second argument is inserted. The problem is that the nested macro call is unmatched within this argument, since its closing delimiter, the newline, lies beyond the end of the argument. ML/I resolves this problem by using the following rule; if, when inserting the Nth argument of a macro call, a construction is unmatched then the Nth delimiter is examined and if this delimiter (or a series of atoms at the start of it) is an exclusive delimiter which closes the apparently unmatched construction then this construction is considered as matched and processing proceeds normally. If there is a nest of unmatched constructions then this rule

is successively applied to all the constructions in turn (in fact, this rule is such a natural one that the user might not realise that there is any logical problem at all).

Note that it is quite legal to insert an exclusive delimiter in the replacement text of the macro to which it belongs. It is even legal to define a name delimiter as an exclusive delimiter (though this is almost certain to lead to an endless loop). Furthermore it is quite legal to have both exclusive delimiters and ordinary closing delimiters within the same delimiter structure.

If a skip ends with an exclusive delimiter, this closing delimiter is not taken as part of the skip and hence it is not affected by the delimiter option associated with the skip.

Exclusive delimiters are sometimes useful in simple applications where no nesting is involved. For instance it is often desirable for a skip to delete up to, but not including, the next newline.

As a more complicated example, consider a language in which macro calls were one to a line with the macro name coming first. In this case it might be convenient to give newline a double use: firstly, as an exclusive delimiter of the macro on the previous line and secondly as a warning marker to precede the macro name on the next line. This is, however, a little tricky; it is usually easier to use the *startline* facility (see Section 3.6 [Startlines], page 25).

The way exclusive delimiters are defined is described at the end of Section 5.1.3 [Introduction to more complicated cases], page 36.

3.6 Startlines *

It is often useful, when processing text where a line is a logical entity (e.g. as in most assembly languages and some high-level languages), to define newline as a macro name. This causes subsidiary problems because

a. The first and last line of the text need to be treated specially.
b. As well as being a macro name, newline may also be a closing delimiter.

To remedy this, ML/I contains an option whereby an invisible layout character called startline may be inserted at the start of each line of input text. The option is controlled by the system variable `S1`: if `S1` is one, startline characters are inserted; if `S1` is not one, they are not. Initially `S1` is zero. ML/I treats startline like any other layout character. Its layout keyword is `SL`.

Startlines are ignored in the output text from ML/I. However, they are not ignored in value text, and users are recommended to set `S1` *after* their macros have been read in. One reason for this is illustrated by the following example:

```
MCDEF TEST OPT ; OR NL ALL
AS <MCGO L1 IF %WD1.=<
>
. . .
```

If `S1` was one while this macro was being read in, then a startline would appear before the `>` character. In this case the test after the `IF`, which should test if delimiter one is a newline, would in fact test if delimiter one was a newline followed by a startline. The test would therefore always fail. If, as is very often the case, startline on its own is a construction name, the above recommendation is virtually imperative.

Example

The following macros would list all labelled statements in an assembly language program. It is assumed the assembler is such that statements are one to a line, and a line is taken to be labelled if the first character is not a space.

```
MCSKIP SL WITH SPACE NL
MCDEF SL NL
AS<%WA1.
>
MCSET S1 = 1
```

3.7 Dynamically generated constructions *

The method of scanning, with the requirement that calls be properly nested, means that all the delimiters of a construction must be in the same piece of text. This rule, which is very desirable since it leads to the early detection of genuine errors, should be borne in mind by the user who wishes to generate constructions dynamically, for example to combine at macro-time separate pieces of text to build up a macro call. The rule prohibits a construction like:

```
CHOOSENAME A TO B;
```

where `CHOOSENAME` is a macro with replacement text `MOVE FROM`, since the call of `MOVE FROM` is not properly nested within the call of `CHOOSENAME`. It is similarly not correct to use the construction:

```
DO A %A1. B REPEAT
```

where `%A1.` has value `TIMES`. It is however quite easy to achieve the object of these examples, namely to generate a delimiter dynamically, and the reader who is interested in doing this should refer to the example in Section 7.4.3 [Dynamically constructed calls], page 78.

4 Operation macros and their use

4.1 Operation macros

The macros considered so far have been strictly concerned with making replacements of pieces of text. In fact, strictly speaking, they should have been called *substitution macros*. There is a second type of macro called an *operation macro*. A call of an operation macro causes a predefined system action to take place, for example the setting up of a new construction. Operation macros are an integral part of ML/I and are not, like substitution macros, defined by the user. They are, however, part of the name environment and are called in the same way as substitution macros. Examples of operation macros are `MCSET` (which performs macro-time arithmetic), `MCDEF` (which defines a macro) and `MCGO` (which is a macro-time conditional 'goto' statement). Examples of their calls are:

```
MCSET P1 = P2+1
MCDEF LNG AS Length
MCGO L6 IF %A1. = ACC
```

Complete descriptions of all the operation macros may be found in Chapter 5 [Operation macros—specifications], page 33. The names of all operation macros begin with `MC` to minimise confusion with substitution macros (users are not forbidden to start their own macro names with `MC`, but it is probably less confusing not to do so). Note that the names of all operation macros, like all other constructions built into ML/I, are written in capital letters.

The arguments of all operation macros are evaluated before being processed. Thus, if `tempno` were a macro with replacement text `P1`, then the following would be equivalent to the previous example of `MCSET`:

```
MCSET tempno = P2+1
```

In most cases, a call of an operation macro does not cause any value text to be generated. No value text would be generated, for instance, in any of the examples above. However, there are two operation macros, `MCSUB` and `MCLENG`, which do cause value text to be generated. These two macros are called *system functions*. `MCSUB` is used for generating substrings of longer pieces of text, and `MCLENG` is used to calculate the length of a piece of text.

There are no general restrictions on the use of operation macros. They may be called from within any type of text, even from within arguments to other operation macro calls.

4.2 Use of literal brackets for surrounding operation macro arguments

The fact that arguments of operation macros are evaluated before being processed has several advantages, but it also has its dangers, and in many cases the user will wish to inhibit this argument evaluation. Consider as an example the last argument of `MCDEF`, which specifies the replacement text of the macro being defined. A definition might be written:

```
MCDEF . . . AS < . . . %A1. . . . >
```

If the above literal brackets (the characters < and >) had been omitted, ML/I would have tried to insert the value of argument one at the time the macro was defined (called *definition time*) rather than when the macro was called, and an error would probably result. Occasionally, however, a user might want to do this, in particular when one macro is defined within another and the arguments of the outer one figure in the definition. Apart from cases like this, it is a good plan to use literal brackets whenever specifying the replacement text of a macro.

Another reason for the usage of literal brackets arises when the replacement text involves one or more newlines, e.g.:

```
MCDEF . . . AS <LINE 1
LINE 2
>
```

In this case, since newline is also the closing delimiter of `MCDEF`, the newlines within the replacement text need to be prevented from closing the `MCDEF`. The literal brackets, being a construction nested within the call of `MCDEF`, achieve this.

It is now possible to see why literal brackets must be defined as matched skips rather than straight skips. Consider the following example, where a piece of replacement text itself contains a call of `MCDEF`:

```
MCDEF MAC1 AS < . . .
        MCDEF MAC2 AS < . . .
        . . . COMMENT > ;
>
```

It is vital that the first < be matched with the last >, and not with the occurrence of > in a comment nor with its occurrence in the nested `MCDEF`. The definition of literal brackets as a matched skip accomplishes this.

4.3 NEC macros

Many of the operation macros have the effect of adding to or deleting from the name environment. These macros are called NEC (name environment changing) macros. The name environment is set up dynamically by calls of NEC macros during text evaluation. The initial state of the name environment is implementation-defined (see Section 2 of the relevant Appendix) but it will usually contain just the operation macros. Changes in the environment affect subsequent text evaluation, but have no effect on value text already generated. Constructions may be defined as either *global* or *local*. Global constructions apply to all subsequent text evaluation, whereas local constructions apply only to the text in which they are defined, together with any macros called from within this text (for exact details see Section 4.4 [Dynamic aspects of the environment], page 29). A local definition occurring in the source text usually has the same effect as a global definition, with the proviso that only the former will be affected by the `MCNO...` operation macros defined in Section 5.2.5 [MCNOWARN MCNOINS MCNOSKIP MCNODEF], page 47.

To start with, most users will probably not be very interested in defining new macros in the middle of text evaluation, In this case, the entire name environment can be set up by a series of NEC macro calls at the start of the source text, and all the rest of the text

can be evaluated using this name environment. Local definition should be used in preference to global ones since the setting up of global definitions involves more work for ML/I (normally, global definitions are only necessary when it is desired to use one macro to set up the definition of another). Readers who are not interested in changing the name environment dynamically can skip Section 4.4 [Dynamic aspects of the environment], page 29, and Section 4.5 [Protected and unprotected inserts], page 30. They can, in fact, totally ignore global definitions, and they need not worry about the difference between protected and unprotected inserts.

4.4 Dynamic aspects of the environment *

The value of a piece of text depends on the state of the environment when its evaluation is started. The purpose of this Section is to define the initial state of the environment when replacement text or inserted text is evaluated, and to explain the effect of dynamic changes in the name environment.

It is convenient to divide the name environment into two parts:

- The *global name environment*, which contains the names of global constructions. Operation macro names are treated as global.
- The *local name environment*, which contains the names of local constructions.

If a substitution macro is called, or if an argument or delimiter is inserted, this cannot change the local name environment of the containing text. However, any change in the global name environment applies to the subsequent evaluation of the containing text. In other words, there is a single global name environment but each piece of text in the process of evaluation has its own particular local name environment.

When a substitution macro is called, the replacement text is evaluated under the following initial environment:

- the global name environment in effect when the call is made.
- the local name environment in effect when the call is made.
- the permanent, character and system variables.
- the arguments and delimiters of the call.
- a set of temporary variables. These are allocated when the call is made. The number allocated is given by the capacity of the macro being called.
- no macro labels.

When an operation macro is called, no special environment is set up and no temporary variables are allocated. The arguments of the operation macro are evaluated under the environment in force when the call was scanned. The same applies to the argument of an insert.

Before considering the initial environment for the evaluation of inserted text, it is instructive to consider an example that will illustrate the reasons behind the rules. This example involves passing arguments down from one macro to another: Assume that within the replacement text of a macro `XYZ` it is desired to call the `MOVE FROM` macro to move the second argument of `XYZ` into a place called `Temp`. This call of `MOVE FROM` would be written:

```
MOVE FROM %A2. TO Temp;
```

This call would cause the replacement text of the `MOVE FROM` macro to be evaluated, and during this evaluation it would be necessary to insert the first argument of `MOVE FROM`. In order to do so, the insert `%A2.` must be performed. Now, in this case ML/I takes `A2` to mean the second argument of `XYZ`, not the second argument of `MOVE FROM`. The initial state of the environment for the evaluation of inserted text is set to make this so. This initial environment consists of:

a. the current global name environment.

b. a local name environment. This depends on whether the insert is protected or unprotected. See Section 4.5 [Protected and unprotected inserts], page 30.

c. the permanent, character and system variables.

d. the arguments and delimiters that were in the environment when the call containing the text to be inserted was encountered.

e. the temporary variables that were in the environment when the call containing the text to be inserted was encountered.

f. no macro labels.

The reader may have noticed that no initial environment contains any macro labels. This is because it is not possible to use the `MCGO` macro to jump from one piece of text to another. Thus each piece of text has its own macro labels, and macro labels are not carried down from one piece of text to another.

4.5 Protected and unprotected inserts *

The difference between protected and unprotected inserts is best illustrated by an example. Consider a macro `ABC` whose replacement text starts as follows:

```
MCDEF Temp AS LMN
%A1.
```

Assume `ABC` is called with `Temp` as its first argument. Then if `%` has been defined as a *protected* insert, the value of `%A1.` is `Temp`. If it has been defined as an *unprotected* insert, the value is `LMN` (note that `MCDEF` defines a local macro). If `MCDEFG`, which defines a global macro, has been used in place of `MCDEF` then the value of `%A1.` would always be `LMN`. Hence the purpose of a protected insert is to protect the insertion of a macro's arguments or delimiters from any changes in the local environment of the macro's replacement text. It is often useful, for instance, to switch into warning mode when entering the replacement text of a macro but still to evaluate its arguments in free mode. In some applications the user may wish to define two insert names, one protected and the other unprotected. In most applications, however, it will be entirely immaterial which sort of insert is defined.

To complete the definition of the previous Section, the initial local name environment when inserted text is evaluated is as follows:

- If the insert is a protected insert then it is the local name environment that was in force when the call containing the inserted text was encountered.
- If the insert is an unprotected insert then it is the local name environment that was in force when the insert was encountered.

4.6 Ambiguous use of names *

When defining new constructions the user should be careful to avoid certain clashes of name. It would obviously be foolish, for instance, to choose the name `MCDEF` for a new construction. ML/I has a fixed set of priority rules for dealing with multiply-defined names, and these are listed below. However, for the reader who is not interested in these complications the following simple rule for defining new constructions is sufficient to avoid difficulty: choose the delimiters to be different from all other environmental names (i.e. the names of macros, inserts, skips, warning markers and stop markers in the current environment). It is quite acceptable, of course, to choose the same representation for the secondary delimiters of different constructions. For example, all macros could have a newline as their closing delimiter. Furthermore it is perfectly in order to have several different names all beginning with the same atom(s); for example three separate macros could have names `RETURN`, `RETURN TO` and `RETURN IF`. ML/I always tries to find the longest name it can, so in this example it would only call the `RETURN` macro if `RETURN` was not followed by `TO` or `IF`. The reader who is prepared to adopt the simple rule above can skip the rest of this Section.

A name clash is considered to occur if an atom or series of atoms of the scanned text can be interpreted in more than one way. Note that some environmental names are ignored within certain contexts (for a complete list, see Section 3.4 [The method of searching for delimiters], page 23); thus a name can sometimes be multiply-defined without a clash occurring. For example, in warning mode it is unambiguous to have a macro name the same as an insert name since each is recognised in a different context.

When a name clash does occur, the following rules are applied in order until all ambiguity is removed:

a. Exclusive delimiters take precedence over everything else.

b. A longer delimiter takes precedence over a shorter one (as illustrated by the above `RETURN` example).

c. Secondary delimiters take precedence over environmental names.

d. Local environmental names take precedence over global ones.

e. The most recently defined environmental name takes precedence.

4.7 Implications of rules for name clashes *

Some implications of the rules are:

- A construction may be overridden by redefining it. It is even possible to redefine a macro within its own replacement text. If it is desired to achieve the effect of deleting a macro name `PQR` from the environment this can be achieved by defining `PQR` as a skip using the `MCSKIP` macro (see Section 5.2.3 [MCSKIP], page 44) as follows:

  ```
  MCSKIP D, <PQR>
  ```

  (`PQR` is enclosed in literal brackets to prevent it being called). This technique can be used for all construction names. Note that when a construction is redefined its old use is not completely deleted (no storage is released) and it is possible under some circumstances to re-incarnate the old usage. For example the overriding use may have

restricted scope or it may be deleted by one of the macros of Section 5.2.5 [MCNOWARN MCNOINS MCNOSKIP MCNODEF], page 47, such as MCNOSKIP.

- It is usually acceptable to choose a construction name to be the same as the secondary delimiter of another construction. For instance, there is no harm in choosing IF as a macro name, even though it is a delimiter of MCGO. The only restriction on the use of IF would be that it could not be called within the first argument of MCGO (this restriction only applies in free mode; in warning mode there would be no restriction).
- A technique (described in Section 7.4.8 [Constructions with restricted scopes], page 82) can be designed to give constructions different meanings in different scopes.
- If it is desired to design a language where each macro call occupies one line, it is practicable to define newline as an exclusive delimiter and also as a warning marker or as a part of a composite macro name (for instance, {*NL*} GOTO could be a macro name). However, it is usually better to use startlines in this kind of situation.
- If each of GO, GO TO, and TO THE END are macro names then:

  ```
  GO TO THE END
  ```

  is interpreted as a call of GO TO, not as a call of GO and a call of TO THE END. This is because the rules of the previous Section are applied at each step in the scan. There is no mechanism for looking ahead and thus deciding, for instance, to take a shorter delimiter at one step in order to get a longer one later.

5 Specification of individual operation macros

This Chapter contains descriptions of the operation macros which should be present in every implementation. In addition, each implementation may have its own particular operation macros (see Section 1 of relevant Appendix).

Arguments of operation macros are evaluated before being processed, in the same way as arguments of substitution macros. Leading and trailing spaces are deleted before evaluation in all cases.

Descriptions of the operation macros have been arranged in a standard format which consists of a number of subsections. These subsections, in order of occurrence, are described below. In some cases, a particular subsection is omitted, if not relevant.

1. *Purpose.*
2. *General form.*
3. *Examples.* Examples may not be comprehensible until further subsections have been read. Each example is independent of all the others.
4. *Restrictions.* This subsection describes any restrictions on the form that the values of the arguments of the macro can take. If this subsection is omitted, there are no restrictions.
5. *Order of evaluation* *. This subsection describes the order in which arguments are evaluated. It is omitted if the order is sequential. The order of evaluation is, of course, immaterial in all but the most pathological cases. Note that any change in the name environment caused by the call of a NEC macro does not come into effect until all its arguments have been evaluated. It is possible for an operation macro to be aborted due to an error before all its arguments have been evaluated.
6. *System action.* This subsection describes the action performed by ML/I at a call of the macro. A reference to the *current environment* means the environment in force when the macro was called. Apart from the system functions, all operation macros have a null value.
7. *Notes.* This subsection contains nothing new, but attempts to bring out more clearly points implied by the preceding material.

Before describing the individual operation macros, it is necessary to describe how to define delimiter structures, since all the operation macros which define new constructions have an argument that specifies the delimiter structure of the construction.

5.1 Specification of delimiter structures

Delimiter structures are defined by writing a *structure representation*, which defines all the delimiters in the structure and the successor(s) of each. The atoms that make up a delimiter are specified by a *delimiter name*, which is written in the following way:

```
{atom} [ ( WITH  ) {atom} *?]
         ( WITHS )
```

The difference between `WITH` and `WITHS` is as follows. If two atoms are linked by `WITHS`, this means that any number of spaces (including none) may occur between the atoms when the delimiter is used. `WITH`, on the other hand, means that no intervening spaces are allowed.

As an example, the delimiter names of a macro of form:

```
COMPARE CHARACTERS {argument 1} /// {argument 2} ;
```

would be:

1. `COMPARE WITHS CHARACTERS`
2. `/ WITH / WITH /`
3. `;`

If, for some reason, it was desired to restrict the number of permissible spaces between `COMPARE` and `CHARACTERS` to one, then this would be specified by:

```
COMPARE WITH SPACE WITH CHARACTERS
```

Note that at least one space must be allowed between `COMPARE` and `CHARACTERS` because otherwise they would not be recognised as separate atoms. Thus, in the general case, a delimiter name is in error if two atoms are connected by `WITH` and neither atom is a punctuation character.

It is now necessary to consider how delimiter names are combined to form a structure representation. In the simplest case, the case of a construction with fixed delimiters, this is done simply by concatenating the delimiter names in the order in which they are to occur. Thus the complete structure representations of some of the constructions used as examples in this manual (see Section 2.10 [Name environment used for examples], page 21) are:

a. `% .`
b. `< >`
c. `COMMENT ;`
d. `DO TIMES REPEAT`
e. `MOVE WITHS FROM TO ;`

5.1.1 Keywords

Within a structure representation the atoms are separated out by *layout characters*, i.e. spaces, newlines, tabs, etc.—in the above examples spaces have been used. Apart from acting as separators, layout characters are totally ignored within structure representations. Thus a problem arises when it is desired to specify a layout character as a delimiter, or as a constituent atom of a multi-atom delimiter. This problem is overcome by using *layout keywords* to stand for layout characters. In particular:

`SPACE`	means a space
`TAB`	means a tab
`SL`	means a startline
`NL`	means a newline
`SPACES`	means a sequence of one or more spaces

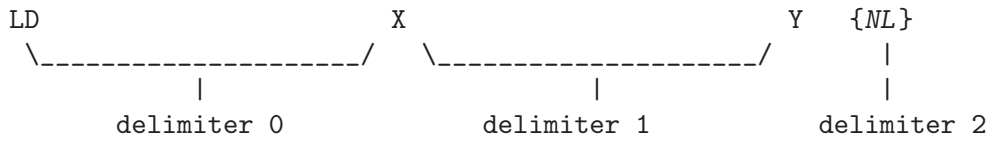
In addition, each implementation may have its own extra layout keywords. See Section 6 of the relevant Appendix for details. The characters represented by these keywords are treated as layout characters and hence, within structure representations, are exactly

equivalent to newlines or spaces. Note that layout keywords only apply within structure representations.

The following are examples of delimiter structures using layout keywords:

a. `ESUB NL`
b. `SPACE`
c. `SL WITH *` (a `*` character at the beginning of a line)
d. `SPACE WITH SPACES` (means two or more spaces)
e. `LD WITH SPACES SPACES NL`

A construction defined using e) above would be analysed thus:



Note how all the spaces following `LD` are absorbed into the name; if they had not been defined to be part of the name they would have been taken as the first delimiter.

It is permissible to use `SPACES` before or after `WITHS`; in these cases it is exactly equivalent to `SPACE`.

In addition to these layout keywords, there are other keywords that apply within structure representations. These are: `WITH`, `WITHS`, `OPT`, `OR`, `ALL` and any atom commencing with the letter `N` followed by a digit. Keywords are reserved words and cannot be used as the atoms of delimiters. However, if it is necessary to define, say, `WITH` as a delimiter name, then the keyword `WITH` could be changed to something else (e.g. `+`) by using the `MCALTER` macro described in Section 5.2.8 [MCALTER], page 50.

5.1.2 The consequences of evaluation

Since structure representations occur as arguments to operation macros, they are evaluated before being processed. Two consequences of this, one beneficial to the user and the other a nuisance, are as follows.

The beneficial consequence is that much-used alternatives can be generated artificially. Assume, for example, that a large number of macros have the form:

```
NAME ( {argument} ) {NL}
```

where `NAME` varies from macro to macro. In this case it would be useful to define a macro `PARENS` with replacement text:

```
WITH ( ) WITH NL
```

Then a macro `DOG` of the above form could be defined by writing:

```
DOG PARENS
```

The mischievous consequence arises if an attempt is made to redefine a macro. Assume that a macro `EMPLOYEE` is defined thus:

```
MCDEF EMPLOYEE AS < J. SMITH >
```

and then subsequently an attempt is made to redefine it by writing:

```
MCDEF EMPLOYEE AS < J. BLOGGS >
```

In this second definition, the structure representation is `J. SMITH` since `EMPLOYEE` is replaced by its value. Hence a macro `J` would be defined with secondary delimiters `.` and `SMITH`. The end result would probably be a puzzling error message, perhaps that a delimiter of the macro `J` was missing.

To avoid problems such as this, it is imperative to enclose a name in literal brackets if it is being redefined. The same applies if the name of one macro occurs as a delimiter of another. In fact, it is not a bad rule to enclose all structure representations in literal brackets except where constructions such as `PARENS` are being used. The correct way to redefine `EMPLOYEE` would be:

```
MCDEF <EMPLOYEE> AS < J. BLOGGS >
```

5.1.3 Introduction to more complicated cases *

The Sections which follow describe facilities for setting up more and more elaborate delimiter structures. Readers are recommended to read on until they know enough for their own applications, and then to skip the rest. Readers who are only interested in fixed delimiters may give up now.

In order to specify the delimiter structure of a construction it is necessary to specify the name(s) of the construction and the successor(s) of each delimiter that is not a closing delimiter. In the simple cases described above, the structure representation consisted of the name of the construction, and then each succeeding delimiter followed by its successor until the closing delimiter. In more complicated cases it is necessary to have two other mechanisms for specifying successors, namely *option lists* and *nodes*. Furthermore it is convenient to imagine that a special symbol `@` occurs at the start of each structure representation, and another symbol • at the end. With this convention any successor of `@` is the name of a construction, and any delimiter with • as a successor is a closing delimiter. The paragraphs which follow contain informal introductions to the concepts of option lists and nodes. More exact details are given in the next Section.

Option lists are used to specify that a delimiter has several optional alternatives as successor. The essential form of an option list is:

`OPT {`*`branch 1`*`} OR {`*`branch 2`*`} OR . . . OR {`*`branch N`*`} ALL`

The ordering of the branches is immaterial. An example of the use of an option list is in the following structure representation for the `ESUB` macro:

```
ESUB OPT TAB OR NL ALL
```

If, in addition, it was decided to allow `SUBTRACT` as an alternative name to `ESUB`, then its structure representation would be:

```
OPT ESUB OR SUBTRACT ALL OPT TAB OR NL ALL
```

In the ordinary way, the successor of the delimiter at the end of a branch is taken as the delimiter following the `ALL` concluding the option list. In other words, the branches may be thought of as coalescing at the delimiter following `ALL` (thus in the example above both `ESUB` and `SUBTRACT` have either tab or newline as alternative successors, and both tab and newline have the imaginary symbol • as successor and are therefore closing delimiters). However, as will be seen, it is possible to override this coalescing effect by the use of nodes.

Nodes are used for defining the successor of a delimiter to be a delimiter or option list elsewhere in the structure representation. The use of nodes in structure representations is analogous to the (much despised) use of labels in programming languages. As the reader will know, the statements in a programming language are written in sequence and the 'successor' of each statement is normally taken as the statement which follows. However, the user can specify a different successor by the use of labels. A label is 'placed' on one program statement and is then 'gone to' after any program statement which requires the labelled statement as successor. In exactly the same way, nodes are used to specify the successors of delimiters.

A node is represented as a *node flag* followed by a positive integer. The normal node flag is the letter `N`, but this can be changed if desired using the `MCALTER` macro of Section 5.2.8 [MCALTER], page 50. It will be assumed in this manual that the node flag is `N`. A node is placed by writing its name before any delimiter name or option list. A node can be 'gone to' only from the end of a branch of an option list or from the end of a structure representation. A 'goto' is indicated simply by placing the name of the appropriate node at the desired point (although the name of a node is used both to place it and to go to it, there is no ambiguity, owing to the different context in which each occurs). As a simple example of the use of nodes, consider the structure representation of a `SUM` macro which allows any number of arguments separated by plus or minus signs and terminated by a semicolon. A typical call of `SUM` would be:

```
SUM A + B - C + D ;
```

The structure representation of `SUM` is:

```
SUM N1 OPT + N1 OR - N1 OR ; ALL
```

This is interpreted thus. `SUM` is followed by either a plus sign, a minus sign or a semicolon. Node `N1` is placed before the option list. The successor of both plus and minus is defined by going to `N1`, and `N1` is associated with the alternatives plus, minus and semicolon. The successor of semicolon, on the other hand, is taken as the delimiter which follows `ALL`, which is •. Hence the semicolon is a closing delimiter.

There are no particular restrictions on the use of nodes. Any number of nodes may be placed within a structure representation provided, of course, that they have different numbers. Any positive integers may be chosen to designate nodes; no particular sequence is required. Node numbers are local to the structure representation in which they occur, and hence there is no relation between the nodes of one structure representation and those of another. Thus the same node numbers may be used in each case. There are no restrictions on the scope of a 'goto'; thus it may dive into an option list or alternatively come out of one.

The node `N0` (`N` zero) has a special usage, namely to denote an exclusive delimiter. Node `N0` may be gone to, but it cannot be placed. If the successor of a delimiter is specified by `N0`, then this delimiter is taken as an exclusive delimiter. Apart from `N0`, it is illegal to go to a node without placing it.

5.1.4 Full syntax of structure representations *

Before describing the general form of a structure representation, it is necessary to describe a number of syntactic sub-components. These are:

a. A *nodeplace* represents the placing of a node, and is specified by the node flag followed by an unsigned positive integer.

b. A *nodego* represents the action of going to a node, and is also specified by the node flag followed by an unsigned integer (in this case, and case a) above, any redundant leading zeros are ignored).

c. A `delspec` represents the specification of a delimiter or an option list, and is of form:

```
[{nodeplace} ?] ({delimiter name}                                    )
                ( OPT {branch} [ OR [{nodeplace} ?] {branch} *?] ALL )
```

where a {*branch*} is of form:

```
{delimiter name} [ {delspec} *?] [ {nodego} ?]
```

(the reader may like to look ahead to the examples in Section 5.1.5 [Examples of complex structure representations], page 39, at this point). Note that each branch must begin with a delimiter name, called the *branch name*. The branch names are the possible alternative successors of the delimiter preceding the option list, and must all be different. Thus no sequence of atoms must match more than one branch name, and the following option list is therefore incorrect:

```
OPT X WITH SPACE WITH Y . . . OR X WITHS Y . . . ALL
```

since `X Y` could be the name of either branch.

As was seen from the preceding example of the `SUM` macro, nodeplaces immediately preceding an option list associate the node with all the options of the list. The syntax forbids a nodeplace immediately after `OPT`, and a nodeplace immediately following `OR` has a special meaning in that it associates the node not only with the delimiter name that follows it but also with the names of all subsequent branches of the option list. As an example, assume that the `SUM` macro was extended to allow the user the option of assigning the answer by writing, for example:

```
SUM X = Y+Z;
```

to calculate `Y+Z` and assign the answer to `X`, or:

```
SUM Y+Z
```

to calculate `Y+Z` and leave the answer in an accumulator.

Here `SUM` has an optional first argument delimited by an equals sign. Its structure representation could be written:

```
SUM OPT = N1 OR N1 + N1 OR - N1 OR ; ALL
```

In this case `N1`, which is placed after the first `OR`, is associated with the alternatives plus, minus and semicolon.

It is also important to ensure that the structure does not become unconnected; for example, the following structure representation is not valid:

```
N1 OPT , N1 OR : N1 All
```

because the structure is a closed loop.

Now that the sub-components have been described it is possible to give the general form of a structure representation. This is:

```
[ {delspec} *]  [ {nodego} ?]
```

5.1.5 Examples of complex structure representations *

This section contains the general forms of some possible constructions, together with the structure representation of each.

Example 1

General form

Either:

```
MEASURE {arg A} METRES {arg B} . {arg C} ;
```

or

```
MEASURE {arg A} YARDS {arg B} FEET {arg C} INCHES {arg D} ;
```

Structure representation

```
MEASURE OPT METRES . OR YARDS FEET INCHES ALL ;
```

In the second form, if it is desired to allow the `INCHES` field optionally to be omitted, then the structure representation could be written:

```
MEASURE OPT METRES . ; OR YARDS
     OPT FEET N1 OR N1 INCHES ; OR ; ALL ALL
```

Here, `N1` is associated with the possibilities `INCHES` and semicolon. In this form the semicolon is mentioned three times. The structure representation could also be written in the following form, where the semicolon only occurs once:

```
MEASURE OPT METRES . N2 OR YARDS
         OPT FEET N1 OR N1 INCHES N2 OR N2 ; ALL ALL
```

(The diagram in the next Section may be an aid to understanding this.)

Example 2

General form

```
[ / {argument} *?] END
```

Structure representation

```
N1 OPT / N1 OR END ALL
```

This macro has two possible names: `/` and `END`.

Example 3

General form

```
(LOAD Q)
(LOAD Q)  {arg A}, {arg B} {NL}
(STORE )
```

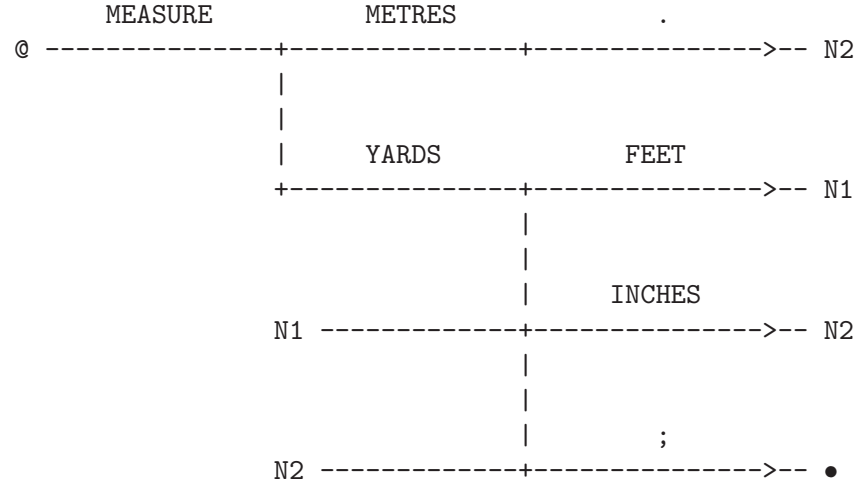
where the newline is an exclusive delimiter.

Structure representation

```
OPT LOAD OR LOAD WITHS Q OR STORE ALL , NL NO
```

5.1.6 Possible errors in structure representations

Great care must be taken in writing structure representations, as errors can have very unfortunate results. In complex cases it may be useful to use a diagram. For example, the following represents the `MEASURE` macro of the previous Section in its final improved form:



Special points to be watched in writing structure representations are the use of keywords and the possible differences between the structure representation as written and its evaluated form. Remember that keywords cannot be used as delimiter names.

If ML/I does reject a structure representation as illegal (giving the message of Section 6.4.6 [Illegal syntax of argument value], page 66), then the following are some of the possible causes:

- Illegal syntax; for example: unmatched `OPT`, node after `OPT`, two nodes in succession, branch without a name, placing of node zero, node names such as `N1A`.
- Keyword used as a delimiter.
- Undefined or multiply-defined node.
- Two branches with the same name.
- Misuse of `WITH` or `WITHS`; e.g. `GO WITH TO`, `X WITHS N1`.
- Structure with no closing delimiter.
- Unconnected structure. For example, the delimiter `D` is not connected to the main structure in the following case:

```
NOGOOD N1 OPT A N1 OR B N1 ALL D
```

5.2 The NEC macros

The operation macros which change the name environment are listed in this Section, one to a page.

In summary, the NEC (name environment changing) macros are:

`MCWARN`	define a local warning marker
`MCWARNG`	define a global warning marker
`MCNOWARN`	delete local warning markers
`MCINS`	define a local insert
`MCINSG`	define a global insert
`MCNOINS`	delete local insert definitions
`MCSKIP`	define a local skip
`MCSKIPG`	define a global skip
`MCNOSKIP`	delete local skip definitions
`MCDEF`	define a local macro
`MCDEFG`	define a global macro
`MCNODEF`	delete local macro definitions
`MCSTOP`	define a stop marker (always local)
`MCALTER`	alter keywords and operation macro delimiters

All of these macros have a null replacement text.

5.2.1 MCWARN

Purpose

Definition of a local warning marker.

General form

`MCWARN {arg A} {NL}`

Examples

a. `MCWARN $`

b. `MCWARN CALL WITHS THIS WITHS MACRO`

Restrictions

`{arg A}` must be a structure representation consisting simply of a single delimiter name.

Notes

If a warning marker is not followed by a macro name, an error message is generated. This message can be suppressed by setting `S3` to a value of one.

System action

`{arg A}` is added to the current environment as a local warning marker, and the current environment is placed in warning mode.

5.2.2 MCINS

Purpose

Definition of a local insert.

General form

`MCINS [ {`*arg A*`}, ?] {`*arg B*`} {`*NL*`}`

Examples

a. `MCINS * .`
defines a protected insert with name `*` and closing delimiter `.`.

b. `MCINS U, INSERT HERE`
defines an unprotected insert called `INSERT`, with closing delimiter `HERE`.

Restrictions

`{`*arg A*`}`, if it exists, must consist of the letter `P` or the letter `U`. Redundant spaces are allowed. `{`*arg B*`}` must be a structure representation of form:

`{`*delimiter name*`} {`*delimiter name*`}`

System action

A new local insert definition is added to the current environment. The delimiter structure of the new insert is represented by `{`*arg B*`}`, and the insert is defined as *protected* unless `{`*arg A*`}` exists and consists of the letter `U`, in which case it is defined as unprotected.

Notes

a. Unprotected inserts are only needed for sophisticated applications of ML/I, and users with simple applications can safely omit `{`*arg A*`}`.

5.2.3 MCSKIP

Purpose

Definition of a local skip.

General form

`MCSKIP [ {`*arg A*`}, ?] {`*arg B*`} {`*NL*`}`

Examples

a. `MCSKIP MT, ( )`
defines `(` and `)` as literal brackets.

b. `MCSKIP N WITH . WITH B WITH . ;`
deletes comments that start with `N.B.` and end with a semicolon.

c. `MCSKIP DT, ' '`

d. `MCSKIP NONL NL`

e. `MCSKIP T, NOPUNCT N1 OPT , N1 OR . N1 OR END ALL`
causes all commas and periods between `NOPUNCT` and `END` to be deleted.

f. `MCSKIP STATIC`
deletes all occurrences of `STATIC`. Note that the delimiter structure of a skip can specify any number of delimiters, although usually there will be one, as in this example, or two.

Restrictions

`{`*arg A*`}`, if it exists, must have the form:

```
[ (M) *?]
  (D)
  (T)
```

Redundant spaces are allowed. `{`*arg B*`}` must be a structure representation.

System action

A new local skip definition is added to the current environment. The delimiter structure of the new skip is represented by `{`*arg B*`}`, and the matched option, the text option and the delimiter option are set if `{`*arg A*`}` contains the letter `M`, `T` or `D`, respectively. If `{`*arg A*`}` is omitted, none of the options are set.

Notes

a. The letters in `{`*arg A*`}` can be in any order.

b. If `{`*arg A*`}` is omitted and `{`*arg B*`}` contains a comma, then this comma should be enclosed in literal brackets to prevent it being taken as a delimiter of `MCSKIP`.

5.2.4 MCDEF

Purpose

Definition of a local macro.

General form

```
MCDEF [ {arg A} VARS ?] {arg B} (AS  ) {arg C} {NL}
                                (SSAS)
```

Examples

a. `MCDEF ARRSIZE AS 6`

b.
```
MCDEF ESUB NL
AS < CMA
     ADD  %A1.
     CMA
>
```
is a definition of the `ESUB` macro used in examples.

c. `MCDEF 6 VARS CALCULATE AS . . .`
defines a macro named `CALCULATE` which has six temporary variables.

d. `MCDEF (OPT + OR - OR * ALL) AS <%D1. %A1. %A2.>`
This macro converts fully parenthesised algebraic notation to Polish Prefix notation. Thus, for example, it would convert

```
((PI * 26)-LENGTH)
```

to

```
- * PI 26 LENGTH
```

e. `MCDEF PARENS AS WITH ( ) WITH NL`
defines the `PARENS` macro used in Section 5.1.2 [The consequences of evaluation], page 35.

f. `MCDEF NOTE ; SSAS < [%WA1.] >`
is the definition of the straight-scan macro `NOTE` used as an example in Section 2.10 [Name environment used for examples], page 21. `SSAS` stands for 'straight-scan `AS`'.

g. `MCDEF CALL NL NO AS . . .`
defines a `CALL` macro with newline as an exclusive delimiter.

Restrictions

{*arg A*}, if it exists, must be a macro expression and {*arg B*} must be a structure representation.

Order of evaluation

{*arg A*}, {*arg C*}, {*arg B*}.

System action

A new local macro definition is added to the current environment. The delimiter structure of this new macro is represented by {*arg B*}, the replacement text is specified by {*arg C*} and the capacity (i.e. the number of temporary variables) is the greater of the result of {*arg A*} and three. The capacity is

three if {*arg A*} is omitted. The new macro is set up as a normal-scan macro if `MCDEF` is called with delimiter `AS`, and as a straight-scan macro if the delimiter `SSAS` is used.

Notes

a. The replacement text is normally enclosed in literal brackets to delay evaluation until macro call time, and to ensure that any newlines within the replacement text are not taken as the closing delimiter of `MCDEF`.

b. If it is desired that the replacement text be treated as a literal when the macro is called as well as when it is defined, then it is necessary to enclose the replacement text in double literal brackets (see example in Section 7.3.1 [Interchanging two names], page 72).

5.2.5 MCNOWARN, MCNOINS, MCNOSKIP and MCNODEF

Purpose

Deletion of local constituents of the current environment.

General form

a. MCNOWARN
b. MCNOINS
c. MCNOSKIP
d. MCNODEF

System actions

These macros respectively delete all local warning markers, all local insert definitions, all local skip definitions and all local macro definitions from the current environment. In addition, MCNOWARN causes the current environment to be placed in free mode unless there are any global warning markers.

Notes

a. Note that these macros do not have newline as a closing delimiter.
b. In current implementations, no storage is released if a constituent of the environment is deleted by one of these macros.
c. See the example in Section 7.3.4 [Deleting a macro], page 73, for a method of deleting individual constructions from the environment.
d. If MCNOWARN is to be meaningful, it must be preceded by a warning marker.
e. MCNODEF does not cause the operation macros to be deleted from the environment since they are global.
f. There is no MCNOSTOP.

5.2.6 MCWARNG, MCINSG, MCSKIPG and MCDEFG

Purpose

Global equivalents of MCWARN, MCINS, MCSKIP and MCDEF.

General form

Similar to those of the corresponding local macros.

Examples

a. `MCWARNG MACRO`
b. `MCINSG /.`
c. `MCSKIPG DT, TEXT : :`
d. `MCDEFG %A1. WITH ( , ) AS < . . . >`

Restrictions

The restrictions on the forms of arguments are the same as for the corresponding local macros.

System actions

As for the corresponding local macros except that the newly-defined constituents are global rather than local.

Notes

a. If a global NEC macro is called in the source text, the effect is the same as if the corresponding local macro had been called (except for certain differences if the name is multiply-defined). Global constructions are not, however, deleted by the macros MCNOWARN etc. described in Section 5.2.5 [MCNOWARN MCNOINS MCNOSKIP MCNODEF], page 47. For reasons of efficiency the user is recommended to use local macros where possible.
b. If a call of MCWARNG occurs, all subsequent text processing will be in warning mode, since it is impossible to delete a global warning marker.

5.2.7 MCSTOP

Purpose

Definition of a stop marker.

General form

`MCSTOP {arg A} {NL}`

Examples

a. `MCSTOP NL`

b. `MCSTOP STOP WITHS RIGHT WITHS HERE`

Restrictions

`{arg A}` must be a structure representation consisting simply of a single delimiter name.

Notes

Stop markers are always local constructions; there is no `MCSTOPG`.

System action

`{arg A}` is added to the current environment as a stop marker.

5.2.8 MCALTER

Purpose

Alteration of the secondary delimiters of operation macros or of the keywords used in structure representations.

General form

`MCALTER {`*arg A*`} TO {`*arg B*`} {`*NL*`}`

Examples

a. `MCALTER`
`TO ;`
`MCALTER AS TO : ;`
After these two calls of `MCALTER`, Example (a), of Section 5.2.3 [MCSKIP], page 44, would be written:

```
MCDEF ARRSIZE : 6;
```

b. `MCALTER WITH TO +`
`MCDEF JOIN + ( WITH ) AS . . .`
`MCALTER + TO WITH`
Here, `WITH` is changed to `+` and then back to `WITH` again in order to define macro `JOIN(` with delimiter `WITH`.

c. `MCALTER N TO 9`
This changes the node flag to the character `9`.

d. `MCALTER SPACE TO BLANK`

Restrictions

`{`*arg A*`}` and `{`*arg B*`}` must be single atoms. `{`*arg A*`}` must be either a secondary delimiter of one or more operation macros, or one of the keywords used in structure representations. `{`*arg B*`}` must not be longer than the system name of any delimiter or keyword matched by `{`*arg A*`}`. If `{`*arg A*`}` is the node flag (i.e. the letter `N` or whatever has replaced it) then `{`*arg B*`}` must be a letter or a digit.

Order of evaluation

`{`*arg B*`}`, `{`*arg A*`}`.

System action

`{`*arg B*`}` is substituted in place of `{`*arg A*`}` wherever `{`*arg A*`}` occurs as a secondary delimiter of an operation macro or as a keyword.

Notes

a. `MCALTER` cannot be used to change the names of operation macros.

b. It is very dangerous to change a keyword or delimiter to become the same as another keyword, for instance:

```
MCALTER UNLESS TO IF
```

The effect of an alteration such as the above on subsequent processing is undefined, since it depends upon the order in which delimiters are scanned.

c. In the unlikely event of a call of `MCALTER` specifying several replacements, some of which are valid, and some of which are invalid because of the length of `{`*arg B*`}`, then the number of valid replacements that are performed before the call is aborted is undefined.

d. In the `MCGO` macro (and in any other macro where the action taken depends upon the form of the delimiters), the delimiters are examined immediately the macro is called and no call of `MCALTER` within an argument can affect the action of the containing macro.

e. Since the operation macros are global, the effect of `MCALTER` is also global.

f. It has been assumed in examples throughout this manual (apart from this Section) that no calls of `MCALTER` have occurred.

g. Since `MCALTER` has a global effect, it is not recommended to use it locally to a piece of replacement text. If it is used locally, `MCALTER` must be called again before leaving the replacement text in order to cancel the changes that have been made.

h. A layout keyword can be `MCALTER`ed to be the same as the character it represents, e.g.:

```
MCALTER NL TO <
>
```

This will effectively delete the layout keyword, e.g. after the above `MCALTER`, newline would stand for itself within structure representations —it would not act as a separator.

5.3 System functions

The operation macros which return values are listed in this Section, one to a page. Note that these macros do not have a newline as the closing delimiter.

In summary, the system function macros are:

`MCLENG`	return the length of a character string
`MCSUB`	return a substring of a character string

The replacement text of `MCLENG` is never null, but the replacement text of `MCSUB` can be anything, including nothing at all.

5.3.1 MCLENG

Purpose

Function to find the length of a character string.

General form

`MCLENG ( {arg A} )`

The left parenthesis is part of the macro name. It may optionally be preceded by spaces.

Examples

a. `MCLENG (%A1.)`

b. `MCLENG(%A1.%D3.PIG)`

System action

The value of this function is the number of characters in `{arg A}`. This number is represented as a character string in the way described in item (e) in Section 2.6.7 [Macro elements], page 14.

5.3.2 MCSUB

Purpose

Function to access a substring.

General form

`MCSUB ( {arg A}, {arg B}, {arg C} )`

The left parenthesis is part of the macro name. It may optionally be preceded by spaces.

Examples

a. `MCSUB (ABC/XYZ, 3, 6)`
This function has value `C/XY`.

b. `MCSUB (ARGUMENT, -2, 0)`
This function has value `ENT`, since non-positive results of {*arg B*} and {*arg C*} specify offsets from the end of {*arg A*}.

c. `MCSUB(%D2.,1,1)`
The value of this function is the first character of the inserted delimiter.

d. `MCSUB (%A3. Y%D3., 1, P3 - T6 + 7)`

Order of evaluation

{*arg A*}, {*arg B*}, {*arg C*}. However, {*arg C*} is not evaluated if `VB` (see below) is greater than `L` (see below) or is less than one.

System action

Let `L` be the number of characters in {*arg A*}, let `RB` be the result of {*arg B*}, and let `VB` be derived from these values by the following rule:

```
          (RB if RB > 0
VB =      (
          (L + RB otherwise
```

Let `VC` be derived from the result of {*arg C*} by a similar rule. The value of a call of `MCSUB` depends upon whether `VB` and `VC` describe a valid substring of {*arg A*}. This occurs if:

```
1 <= VB <= VC <= L
```

If this relation does not hold, the value of `MCSUB` is null. If the relation holds, the value of `MCSUB` is the substring of {*arg A*} from character position `VB` up to and including character position `VC`, the first character of {*arg A*} being taken as character position one.

Notes

a. In the case where the relation holds, the value of `MCSUB` consists of `VC-VB+1` characters.

b. The value of `MCSUB` is not itself evaluated. Thus the value of example b) above would be `ENT` even if `ENT` was a macro.

5.4 Further operation macros

The remaining operation macros, i.e. those not falling into the previous categories, are described below.

In summary, they are:

`MCSET`	macro-time assignment statement
`MCNOTE`	generate error and debugging messages
`MCGO`	macro-time 'goto' statement
`MCPVAR`	allocation of permanent variables
`MCCVAR`	allocation of character variables

All of these macros have a null replacement text, although `MCNOTE` will generate output on the debugging file.

5.4.1 MCSET

Purpose

Macro-time assignment statement.

General form

`MCSET {arg A} = {arg B {NL}`

Examples

a. `MCSET P10 = 3`

b. `MCSET T6 = -4`

c. `MCSET TT3 = TP4 - 109 + 25/P1`

d. `MCSET T%A1. = %A1. + 17`

where the value of the inserted argument is a positive integer.

Restrictions

`{arg A}` must be the name of a macro variable in the current environment (`{arg A}` may contain redundant spaces at the beginning or the end). `{arg B}` must be a macro expression if `{arg A}` describes an integer macro variable, or an arbitrary character string (of length no greater than the current range) if `{arg A}` describes a character macro variable.

System action

The result of `{arg B}` is assigned to the macro variable designated by `{arg A}`.

Notes

a. When values are assigned to character variables, the whole of the second argument to `MCSET` (after the usual deletion of spaces at the beginning and end of the argument) is assigned to the specified character variable. If leading or trailing spaces are to be preserved, the argument must be enclosed in literal brackets. For example:

```
MCSKIP MT,<>
MCSET C1 =       ABC        length of contents of C1 is three
MCSET C1 = <     ABC    >   length of contents of C1 is ten
```

b. No particular operators are provided for the manipulation of character variables, since the basic ML/I facilities are sufficient. To add together (concatenate) two character variables, simply insert both of them:

```
MCSET C1 = %C2.%C3.
```

5.4.2 MCNOTE

Purpose

Generation of user's own error and debugging messages.

General form

`MCNOTE {arg A} {NL}`

Examples

a. `MCNOTE %A3. is illegal argument`

b. `MCNOTE Occurrence number %P1. of <CONT>`

System action

`{arg A}` is printed on the debugging file (see Chapter 6 [Error messages], page 63) as if it were a system message. A newline is inserted in front of it, and it is followed by a printout of the context of the call of `MCNOTE`.

Notes

a. If example b) occurred in line 3 of a macro `CONT`, then the output might be:

```
Occurrence number 33 of CONT
detected in
line 3 of macro CONT with no arguments
called from
line 267 of source text
```

b. Notes d) and f), of Section 6.2 [Notes on context print-outs], page 63, do not apply to the printing of `{arg A}`.

5.4.3 MCGO

Purpose

Macro-time 'goto' statement or conditional 'goto' statement.

General forms

a. `MCGO {arg A} {NL}`

b. `MCGO {arg A} (IF    ) {arg B} (= ) {arg C} {NL}`
`(UNLESS)          (BC)`
`(EN)`
`(GE)`
`(GR)`

The meanings of the respective mnemonic second delimiters are: Belongs to Class, Equals Numerically, Greater than or Equals, and GReater than.

Examples

a. `MCGO L1`

b. `MCGO LT1`

c. `MCGO L6 IF %D1. = +`

d. `MCGO L0 UNLESS P3 - T5 GE - 6`

e. `MCGO L T3 - P7 + 4 UNLESS %A6. BC N`
This tests whether argument six is a number (Belongs to the Class of Numbers).

Restrictions

`{`*arg A*`}` must consist of the letter `L` (optionally preceded by redundant spaces) followed by a macro expression. The result of this macro expression must never be negative and, furthermore, it must not be zero if `MCGO` is called from the source text. If the second delimiter is `BC` then `{`*arg C*`}`, which is the name of a class, must consist of one of the following letters:

`I` (for identifier)

`L` (for letter)

`N` (for number)

together with any desired number of spaces. If the second delimiter is `EN`, `GE` or `GR` then `{`*arg B*`}` and `{`*arg C*`}` must both be macro expressions.

Order of evaluation

`{`*arg B*`}`, `{`*arg C*`}`, `{`*arg A*`}`. In form b), `{`*arg A*`}` is evaluated only if the condition holds.

System action for form b)

`{`*arg B*`}` and `{`*arg C*`}` are compared to yield a true or false value. If the second delimiter is `EN`, `GE` or `GR`, then numerical comparison is performed; otherwise character comparison is performed. The method of comparison depends on the second delimiter in the following way:

`=` A true value results only if {*arg B*} and {*arg C*} are identical strings of characters.

`BC` If {*arg C*} is the letter `I`, then a true value results only if {*arg B*} is of form:

```
[ {letter} *]
  {digit }
```

If {*arg C*} is the letter `L`, then a true value results only if {*arg B*} is of form:

```
[ {letter} *]
```

If {*arg C*} is the letter `N`, then a true value results only if {*arg B*} is of form:

```
[ (+) *?] [ {digit}]
  (-)
```

`EN, GE, GR`

In these cases a true value results only if the result of {*arg B*} is, respectively, numerically equal to, greater than or equal to, or greater than the result of {*arg C*}.

If the comparison yields a false value and the second delimiter is `IF`, or if the comparison yields a true value and the second delimiter is `UNLESS`, then no further action takes place. Otherwise the system action for form a) is now performed.

System action for form a)

Let `N` be the result of the macro expression in {*arg A*}. If `N` is positive, then the point of scan is changed to the point associated with macro label `N` (see below for a fuller description). If `N` is zero, then processing of the current piece of text is abandoned and evaluation proceeds as if the end of the current piece of text had been reached. Thus when `N` is zero a `MCGO` serves a similar function to the `RETURN` statement found in many high-level languages. This 'return' facility may be used within inserted text or replacement text but not within the source text.

Notes

a. Note that leading and trailing spaces are removed before {*arg B*} and {*arg C*} are evaluated. If it is required that these spaces take part in the comparison, they should be enclosed in literal brackets.

b. If it is desired to achieve the effect of a backward 'goto' in the source text, then the required loop must be defined as the replacement text of a macro call. For an example, See Section 7.4.1 [Macro-time loop], page 76.

c. Section 7.4.9 [Optimising macro-generated code], page 83, as well as Section 7.3.5 [Differentiating special-purpose registers and storage locations], page 73, contain examples of the use of `MCGO`.

d. The user should be very careful to differentiate between the two relational operators `=` and `EN`. Note that the relation:

```
P1 EN P2
```

is true if the first two permanent variables have the same value, whereas:

```
P1 = P2
```

is, of course, never true. Note that:

```
%P1. = %P2.
```

is equivalent to:

```
P1 EN P2.
```

*Exact description of a 'goto' **

The following is a more exact description of the action of ML/I in performing a 'goto' when `N` is positive.

If label `N`, which is called the *designated label*, is present in the current environment then the action of ML/I is simply to change the point of scan to the point associated with the designated label. Otherwise a forward search for the designated label is performed, starting at the current point of scan. If a macro call or skip is encountered during this search, the search is suspended until the end of the macro call or skip is found. Each time an insert is encountered outside a call or skip, the argument is evaluated and the search ends when an insert which 'places' label `N` is found (or, in the error case, at the end of the current piece of text). No value text is generated during a search and no macro calls are performed (except conceivably during the evaluation of the argument of an insert). At the end of the search the action of ML/I is concluded by setting the point of scan as the point immediately after the designed label.

Any labels encountered in the forward search (including the designated one) are added to the current environment provided that it is possible to satisfy the rules of part f) of Section 2.6.7 [Macro elements], page 14. If an error is detected during a forward search then the appropriate error message is output in the normal way.

5.4.4 MCPVAR

Purpose

Allocation of extra permanent variables.

General form

`MCPVAR {arg A} {NL}`

Examples

a. `MCPVAR 100`

b. `MCPVAR T1+3`

Restrictions

`{arg A}` must be a macro expression.

System action

Let `N` be the result of `{arg A}`. If `N` is greater than the current number of permanent variables, then the number of permanent variables is increased to `N`; otherwise no action is taken. The values of the new permanent variables are set to zero, and the values of the previously allocated ones remain unchanged.

5.4.5 MCCVAR

Purpose

Allocation of extra character variables.

General form

`MCCVAR {arg A} [ , {arg B} ?] {NL}`

Examples

a. `MCCVAR 50`

b. `MCCVAR T1*4`

Restrictions

`{arg A}` and `{arg B}` must be macro expressions.

System action

Let `N` be the result of `{arg A}`, and let `M` be the result of `{arg B}`.

If this is the first call of `MCCVAR` in the current process, `{arg B}` must be present, and the current range is set to `M`. If this is not the first call of `MCCVAR`, `{arg B}` may be omitted; however, if `{arg B}` is present, `M` must equal the current range (in other words, the range may not be altered once it has been set).

If `N` is greater than the current number of character variables, then the number of character variables is increased to `N`; otherwise no action is taken. The values of the new character variables are set to null (the empty string), and the values of the previously allocated ones remain unchanged.

6 Error messages

ML/I detects all errors and prints a message at every occurrence. An error message consists of a statement describing the particular error that has been detected, with a print-out of the current context. This print-out enumerates all the macro calls and insertions of arguments or delimiters that are currently being processed, together with a line number to indicate the state of the scan in each case. Error messages are printed on an implementation-defined medium (see Section 4 of relevant Appendix) called the *debugging file*. This is normally an interactive display, or a separate output file.

6.1 Example of an error message

An example of an error message is the following. Assume the user has written:

```
MCSET Y10 = 56
```

in the source text. Then the following message would be given:

```
Error(s)
Argument has illegal value, viz "Y10"
detected in
macro MCSET with arguments
1)   Y10
2)   56
called from
line . . . of source text
```

6.2 Notes on context print-outs

The print-out of the context should be largely self-explanatory, but the following points should be noted.

a. The line number is one greater than the number of newlines so far encountered in the piece of text to which it refers. Line numbers refer to scanned text, not to value text.

b. If a macro call or insert straddles more than one line of text, then the line numbers of both the beginning and the end of the call or insert are printed, e.g.:

```
called from lines 6 to 21 of source text
```

c. When the arguments of a call are enumerated, the text of each argument rather than its value is printed.

d. If a piece of text in an error message consists of a single layout character, then the corresponding layout keyword, enclosed in parentheses, is used in its place, for example:

```
Delimiter (NL) of macro X not found
```

In addition a null piece of text is represented by `(NULL)`.

e. Any multi-atom delimiter occurring in an error message is printed in full. A space is printed between two adjacent atoms if spaces are permitted between the atoms (i.e. if `WITHS` has been used rather than `WITH` in their definition). Note d) above applies to each atom. As an example, a message involving the multi-atom macro name `MCSUB (` would read:

```
Macro MCSUB ( called from . . .
```

f. There is an implementation-defined number 2N (see Section 4 of the relevant Appendix) which is the maximum length of a piece of text that can be inserted in an error message. If a piece of text is too long, the first N−4 characters and the last N−4 characters are printed, separated by three dashes and some spaces.

g. If the text of an error message is about to overflow a line, then a newline is artificially inserted.

6.3 Count of errors

If ML/I detects an error during processing, its normal action is to display the error message and then continue. There are many cases where it is useless to continue, so a count of errors is available in `S5` (to be precise, `S5` is a count of the number of times that the message prologue `Error(s)` has been output on the debugging file).

Macro packages can test the value of `S5` at regular intervals (for example, at the start of each line) and can abort the process in some implementation-dependent way if `S5` has passed some threshold. `S5` can be assigned to like any other macro variable—all ML/I does to it is increment its value by one at each error.

6.4 Complete list of messages

This Section contains a complete list of all the error messages produced by ML/I; it also includes other messages which are merely informational.

6.4.1 Illegal macro element

Message

`{`*flag*`} {`*number*`} is illegal macro element`

Description

The `{`*number*`}`, which is the value of the subscript or macro expression associated with the `{`*flag*`}`, is either too large or too small. Alternatively, macro elements of the type designated by the `{`*flag*`}` do not exist in the current environment (e.g. there are no arguments or temporary variables in the source text).

System action

The current operation macro or insert is aborted.

6.4.2 Arithmetic overflow

Message

`Arithmetic overflow`

Description

Overflow has occurred during the evaluation of a macro expression or subscript. This message occurs when an attempt is made to divide by zero. It may also occur under other circumstances, but these are implementation-defined (see Section 5 of relevant Appendix).

System action

The current operation macro or insert is aborted.

6.4.3 Illegal input character

Message

`Illegal input character`

Description

A character of the source text is not in the character set of the implementation.

System action

The illegal character is replaced by a fixed implementation-defined character called the *error character* (see Section 4 of relevant Appendix). A typical error character is the question mark.

6.4.4 Illegal macro name

Message

`Illegal macro name after warning, viz "{`*atom*`}"`

Description

A warning marker is followed, possibly with intervening spaces, by the given `{`*atom*`}` which is not a macro name (nor the start of a multi-atom macro name). If this error occurs within an argument, the above message is printed both when the argument is originally scanned and also each time it is inserted.

System action

The warning marker is treated as if it had not been recognised as an environmental name, and the atom which follows is treated as if no warning marker had occurred. Thus, for example, a skip name following a warning marker will be treated as a skip name.

This message is suppressed altogether if system variable `S3` is set to one (see Section 8.2 [Use of S1 to S9], page 85).

6.4.5 Unmatched construction

Message

```
Delimiter {name} [or {name} *?] of (macro ) {name}
                                   (skip  )
                                   (insert)
in line {number} of current text not found
```

Description

The given construction which starts in the given line of the current piece of text is not complete. Note that the line number is relative to the current piece of text. When the error was detected the scan was searching for the given delimiter (or for one of the given alternative delimiters). The error is detected only when the scan reaches the end of the source text or the end of a piece of inserted text or replacement text, or a stop marker is encountered.

Possible causes

A mismatch of the delimiters of a construction nested within the given one can cause this error since delimiter matching is liable to get out of phase as a result. Alternatively, an incorrect specification of a delimiter structure can cause delimiters to be matched in a way not intended by the user and, again, the error may be in a nested construction rather than in the given one.

System action

In the call and insert cases, the effect is as if the text from the macro or insert name to the current point of scan was deleted. In the skip case, text skipped over is treated in the normal way and the skip is artificially terminated.

6.4.6 Illegal syntax of argument value

Message

`Argument {`*`number`*`} has illegal value, viz "{`*`value`*`}"`

Description

The given value of an argument to an operation macro or insert has not the required syntax. For operation macro arguments see appropriate 'Restrictions' subsection of Section 5.2 [The NEC macros], page 41, Section 5.3 [System functions], page 52, or Section 5.4 [Further operation macros], page 55, or if the argument is (supposed to be) a structure representation then see Section 5.1.6 [Possible errors in structure representations], page 40. For arguments to inserts, see Section 2.6.7 [Macro elements], page 14.

System action

The current operation macro or insert is aborted.

6.4.7 Redefined label

Message

`Label {`*`number`*`} is multiply-defined`

Description

An attempt has been made to re-define a label that has already been defined within the current text.

System action

The new definition is ignored.

6.4.8 Undefined label

Message

`Label {`*`number`*`} referenced in line {`*`number`*`} of current text not found`

Description

A call of `MCGO` references an undefined label. This error is detected when the scan reaches the end of a piece of text (since it performs a search for the missing label). If any constructions are unmatched, the relevant message(s) (of Section 6.4.5 [Unmatched construction], page 65) are printed with this message.

Possible causes

An attempted backward `MCGO` in the source text or an attempted `MCGO` from one piece of text to another can cause this error. Alternatively, it can be caused by an unmatched construction within the scope of a forward `MCGO`.

System action

The effect is as if the designated label had been found at the very end of the current piece of text.

6.4.9 Storage exhausted

Message

```
Process aborted for lack of storage
[possibly due to {other messages} ?]
```

Description

ML/I has used up all its available storage. If the current text is the source text then the following additional information is given: if there are any constructions currently unmatched, or if a search is being made for a macro label as the target of a `MCGO`, one or more of the relevant messages (of Section 6.4.5 [Unmatched construction], page 65, and Section 6.4.8 [Undefined label], page 67) are printed with this message.

Possible causes

Storage is taken up by macro variables, by the name environment, by a macro call or insert in the source text, and by nested calls and/or inserts. Hence an unmatched macro call in the source text or a call with a very long argument can cause this error. Alternatively, it can be caused by an endless or very deep recursive nest, by the name environment being too big, or by a combination of all these factors.

System action

The current process is aborted.

6.4.10 System error

Message

`System error {`*`number`*`}`

Description

There has been a machine error, an operating error or an error in the implementation of ML/I. The value `{`*`number`*`}` indicates the location within the ML/I logic where the error was detected, and in general will be useful only when reporting the error to the implementor.

System action

The current process is aborted.

6.4.11 Subsidiary message

Message

`(Macro ) {`*`name`*`} aborted due to above error`
`(Insert)`

Description

This message occurs as a subsidiary message every time an error causes the operation macro or insert currently being performed to be aborted. Any construction that has been aborted is given a null value.

6.4.12 Statistics

Typical

`At end of process: {`*`number`*`} lines, {`*`number`*`} calls`

Description

The occurrence of this message is implementation-defined (see Section 4 of relevant Appendix). It is usually output at the end of a process and sometimes at intermediate stages as well. The number of lines of source text that have so far been scanned, together with the total number of macro calls performed (the value used as an initial setting of `T2`) is output.

6.4.13 Version number and current constructions

Message

```
Version {version-string}
 .     .
Stops are
 .     .
Macros are
 .     .
Warnings are
```

```
 .     .
Inserts are
 .     .
Skips are
 .     .
```

Description

It is implementation-defined whether this message is output by default. {*version-string*} is the version of the machine-independent logic of ML/I. The version message is followed by a list of all the currently defined constructions, grouped by type.

This message is purely informational, but may be useful in identifying certain problems.

6.4.14 Implementation-defined messages

Description

Each implementation may have its own particular messages. See Section 4 of the relevant Appendix for details.

7 Hints on using ML/I

7.1 How to set up the environment

It is best, where possible, to write all the NEC macro calls to set up the environment at the start of the source text (on many implementations, there is a facility for using multiple input files, so these can be placed into a separate 'prologue' file). The name environment will normally contain an insert definition, and it is a good idea to define this first. Choose some atom(s) as the insert name that will not occur naturally in the source text to be processed. Also define a pair of literal brackets, again choosing atoms that do not occur naturally in the source text. Thus do not use `<` and `>` if these symbols are used to represent 'less than' and 'greater than'. Finally, define the required macros, not forgetting to enclose arguments in literal brackets where necessary. It may be useful to have a systematic convention for macro names, for example starting every macro name with the same letter. However, due to the randomising technique used in the internal working of many implementations of ML/I, it is not advisable to choose macro names all of the same length and all ending with the same character, as this would slow down execution.

7.2 Possible sources of error

The following Sections illustrate some areas where the user of ML/I should take special care.

7.2.1 Jumping over expanded code

If macros are used in an assembly language, great care must be taken with instructions of the form 'jump to location counter `+ N`', since there may be macros within the scope of the jump which expand into several machine instructions. The same applies to machine instructions of the form 'skip one instruction'. For this reason it is helpful to choose macro names that cannot be confused with the names of machine instructions.

7.2.2 Generation of unique labels

If a macro generates code which involves an execution-time label, then a different label must be generated at each call of the macro. The technique described earlier, in part b) of Section 2.6.8 [Insert definitions], page 14, can be used for this purpose. The same applies, in some cases, to execution-time temporary variables.

7.2.3 Lower case letters

Note that only upper case letters may be used for vocabulary words of ML/I. This applies to the names and secondary delimiters of operation macros, to keywords and to insert flags. Further note that, for example, `PIG`, `Pig` and `pig` are three different atoms.

7.2.4 Use of newlines in definitions

Remember that layout characters within replacement text are treated like any other characters. They should therefore be used with great care as they affect the format of the output text. Thus:

```
MCDEF LOAD AS <LD>
    LOAD  X
```

would generate:

```
    LD  X
```

whereas:

```
MCDEF STORE
AS  <ST
>
    STORE  Y
```

would generate:

```
    ST
  Y
```

Moreover:

```
MCDEF JUMP AS
<B>
    JUMP  LB6
```

would generate:

```
B
      LB6
```

since `JUMP` would be defined as a null macro.

7.2.5 Use of redundant spaces

As a general rule, extra spaces are ignored within text that forms an instruction to ML/I, but are treated like any other character within text that ML/I manipulates.

Spaces may be chosen as construction names, but, in any context where spaces are ignored, they are ignored even if space is a construction name. In particular, spaces are ignored after warning markers so, when in warning mode, it is not possible to have a macro name commencing with a space.

Below is a list of some of the places where spaces are ignored:

a. At the beginning or end of an argument to an operation macro (before evaluation).
b. Ditto for an argument to a substitution macro, provided the insert flag `B` is not used.
c. After a warning marker.
d. Within a macro expression (except within variable names or constants).
e. Within the argument to an insert (except within variable names or constants).
f. Within the values of those operation macro arguments that specify options. Within structure representations, one or more spaces act as a separator.

7.3 Simple techniques

This Section illustrates a few techniques for solving some simple problems. In general, only one solution is given but there are often several equally good solutions. In some cases a problem has been described in terms of the use of ML/I as a preprocessor to a particular language, but in each case the problem has counterparts in other applications.

7.3.1 Interchanging two names

Problem

It is desired to interchange the names `PIG` and `DOG` in a piece of text.

Solution

The complete name environment is set up as follows:

```
MCSKIP MT, < >
MCDEF PIG AS <<DOG>>
MCDEF DOG AS <<PIG>>
```

and the desired result is achieved by evaluating the given text under this environment.

Notes

a. In this example there is no necessity to have an insert definition in the environment.

b. Notice that two pairs of literal brackets are used to surround the pieces of replacement text. One pair is stripped off at definition time and the second at replacement time. If the brackets were omitted, ML/I would endlessly replace `PIG` by `DOG` by `PIG` by `DOG` . . .

7.3.2 Removing optional debugging statements

Problem

It is desired to include a number of extra statements in a FORTRAN program in order to aid in debugging its execution. These are to be removed when the program is debugged. Each statement ends with a newline.

Solution

Some unique atom, say `DEBUG`, is written at the beginning of each debugging statement. Before the FORTRAN program is compiled it is passed through ML/I. If it is desired to include the debugging statements, then the following skip definition is placed in the name environment:

```
MCSKIP DEBUG
```

This causes each occurrence of `DEBUG` to be deleted. When it is desired to delete the debugging statements then the following skip definition is used:

```
MCSKIP DEBUG NL
```

If `DEBUG` is always to be at the beginning of a line, it may be better to define the skips with:

```
MCSKIP SL WITH DEBUG
```

and

```
MCSKIP SL WITH DEBUG NL
```

as this will prevent matches with `DEBUG` if it occurs elsewhere in the scanned text.

7.3.3 Inserting extra debugging statements

Problem

It is desired in an X123 Assembly Language program to replace every occurrence of `DAC COW` (deposit accumulator at `COW`) by a call to a subroutine (which perhaps prints the value assigned to `COW`). This call has form `JMS TYPCOW`.

Solution

```
MCDEF DAC WITHS COW AS <JMS TYPCOW>
```

7.3.4 Deleting a macro

Problem

It is desired to delete the macro `GONE` from the current environment.

Solution

The following skip accomplishes this:

```
MCSKIP D, <GONE>
```

Notes

a. The literal brackets prevent `GONE` being called during the evaluation of the second argument of the above `MCSKIP`.

b. Strictly speaking, the macro `GONE` is overridden rather than deleted (see part a) of Section 4.7 [Implications of rules for name clashes], page 31.

7.3.5 Differentiating special-purpose registers and storage locations

Problem

It is desired to define an `INTERCHANGE` macro for X123 Assembly Language so that, as well as being used to interchange the values of two storage locations, it can be used to interchange the accumulator with a storage location. In the latter case `ACC` is written as the first argument of the call.

Solution

Assuming the existence of a `MOVE FROM` macro, which moves the value of one storage location into another, the definition of `INTERCHANGE` is written:

```
MCSKIP " WITH " NL
MCDEF INTERCHANGE WITH ( , ) WITH NL
AS <MCGO L1 IF %A1. = ACC
MOVE FROM %A2. TO TEMP;MOVE FROM %A1. TO %A2.;
MOVE FROM TEMP TO %A1.;
MCGO L0
%L1.     DAC TEMPAC        "" deposit accumulator in TEMPAC
MOVE FROM %A2. TO TEMP;MOVE FROM TEMPAC TO %A2.;
     LAC TEMP              "" load accumulator from TEMP
>
```

Note the use of the initial skip here, to provide a comment facility.

7.3.6 Testing for macro calls

Problem

It is desired to find out whether an argument of a macro call itself involves any macro calls, inserts or skips.

Solution

Compare the written form of the argument with its evaluated form (it is assumed that any construction occurring within the argument would cause these two forms to be different). The following is an example of how the test might be written:

```
MCGO L1 IF %A1. = %WA1.
```

Alternatively, if it was only required to test if the argument involved any macro calls, the test might be written:

```
MCGO L1 IF MCNODEF%A1. = %A1.
```

provided that `%` had been defined as an *unprotected* insert.

7.3.7 Searching

Problem

It is desired to search the source text to find all occurrences of given atoms.

Solution

Define macros such as:

```
MCDEF X
AS <MCNOTE HERE IS <X>
>
```

It is best to send the output text itself to a null output file so that the only printed output is the `MCNOTE` messages.

7.3.8 Bracketing within macro expressions

Problem

Parentheses cannot be used within macro expressions.

Solution

Use nested inserts. For example to insert the value of:

```
(P1+6)/(P3-2)
```

write:

```
%%P1+6./%P3-2..
```

7.3.9 Deletion from source text only

Problem

It is desired to delete a given atom only if it occurs in the source text.

Solution

Use temporary variable three, e.g.:

```
MCDEF X AS <MCGO L0 IF T3 EN 1
%WD0.>
```

7.3.10 Locating missing delimiters

Problem

A missing delimiter may cause ML/I to 'run away' while scanning, perhaps causing it to exhaust the available storage. Although the error messages will locate the problem, they are likely to be excessive.

Solution

Define a stop marker. Experience with stop markers has shown that, in nine out of ten applications of ML/I, it is a good idea to include:

```
MCSTOP NL
```

in the environment. It is best to make this the last definition, since calls of certain operation macros may legally straddle several lines.

7.3.11 Handling line-oriented input

Problem

ML/I treats its input as 'free format', whereas a particular piece of input is line oriented and therefore difficult to handle.

Solution

Use startlines. For example, suppose that it is desired to handle labelled and unlabelled statements in X123 Assembly Language separately. Assume that statements are one per line, and that unlabelled statements begin with at least one space, but that labels must occur at the start of a line. The macro to handle unlabelled statements would look like this:

```
MCDEF SL WITH SPACES . . . NL AS . . .
```

and the macro to handle labelled statements could be defined as:

```
MCDEF SL . . . NL AS . . .
```

remembering, of course, to turn startlines on (see Section 8.2 [S1 to S9], page 85) before scanning the actual X123 program. Note that although both macro names begin with the atom `SL`, ML/I will always try to find the longest match, so there is no danger that an unlabelled statement will cause a call of the macro intended for labelled statements.

Notes

It is possible to achieve a similar effect by using macros starting with a newline, with the closing delimiter being another newline as an exclusive delimiter. However, this is rather more tricky than using startlines.

7.4 Sophisticated techniques *

This Section illustrates some techniques which may be of value to the more sophisticated user.

7.4.1 Macro-time loop

Problem

A macro-time iteration statement is required in order to generate repetitive text.

Solution

The macro `MCFOR` defined below serves this purpose. It allows the step size to be optionally omitted; in this case a step size of one is assumed. `MCFOR` should be regarded as a 'black box' by the reader who finds the definition below hard to understand. The part labelled by `L2` is to deal with a negative step size.

```
MCDEF MCFOR = OPT STEP N1 OR N1 TO ALL NL REPEAT
AS<MCSET %A1. = %A2.
MCSET T3 = 1
MCGO L1 IF T1 EN 4
MCSET T3 = %A3.
MCGO L1 IF T3 GR 0
%L2.MCGO L0 IF %AT1-1. GR %A1.
%AT1.MCSET %A1. = %A1. + T3
MCGO L2

%L1. MCGO L0 IF %A1. GR %AT1-1.
%AT1.MCSET %A1. = %A1. + T3
MCGO L1
>
```

Examples

a. To generate the twenty instructions:

```
JMP LAB1
JMP LAB2
  . .
  . .
JMP LAB20
```

one could use:

```
MCFOR P1 = 1 TO 20
JMP LAB%P1.
REPEAT
```

b. To generate the above twenty instructions in reverse order:

```
MCFOR P6 = 20 STEP -1 TO 1
JMP LAB%P6.
REPEAT
```

c. To generate the first ten powers of two:

```
MCSET P2 = 1
MCFOR P1 = 1 TO 10
%P2.MCSET P2 = P2+P2
REPEAT
```

Notes

a. The controlled variable must be a permanent variable (if it were a temporary variable, `MCFOR` would try to use its own temporary variables rather than those of the calling environment thus causing an error).

b. The initial value, step size, and final value must be macro expressions not involving temporary variables.

c. `MCFOR` is a substitution macro, not an operation macro.

d. Calls of `MCFOR` may be nested.

e. `MCFOR` can be used to perform loops within the source text, thus surmounting the restriction that source text 'gotos' are not allowed.

7.4.2 Examining optional delimiters

Problem

An `IF` macro has form:

```
IF {arg A} (GE) {arg B} THEN . . .
           (GR)
           (LT)
           (= )
           (etc.)
```

Within the replacement text of `IF`, it is desired to examine the form of the first delimiter and go to `L1` if the delimiter is `GE`, to `L2` if it is `GR`, etc. This problem can obviously be solved by writing a large number of conditional `MCGO` statements but this would make the `IF` macro very slow and cumbersome.

Solution

The various possible delimiters can be defined as macros thus:

```
MCDEF GE AS 1
MCDEF GR AS 2
    etc.
```

and then the requisite switch statement can be written:

```
MCGO L%D1.
```

Notes

a. The definition of the delimiters of `IF` as macros does not affect the scanning of a call of the `IF` macro since the use of an atom as a delimiter takes precedence over its use as a macro name.

b. It is necessary to place the definitions of `GE` etc. after the definition of `IF` or else to enclose the structure representation of `IF` within literal brackets.

c. The technique will not, as it stands, work for name delimiters. However, see Section 7.4.8 [Constructions with restricted scopes], page 82).

7.4.3 Dynamically constructed calls

Problem

It is required to implement a `WHILE` macro of form:

```
WHILE {arg A} (GE) {arg B} DO
              (GR)
              (LT)
              (= )
              (etc.)
{arg C}
END
```

Within the replacement text of this macro it is desired to call the `IF` macro with the first delimiter of this call of `IF` the same as the delimiter that occurred in the call of `WHILE`. However, as was seen in Section 3.7 [Dynamically generated constructions], page 26, it is not possible to do this by writing:

```
IF . . . %WD1. . . . THEN . . .
```

Solution

It is necessary to use a temporary macro definition to build up the text for the required call of `IF` and then to call the temporary macro. This could be achieved thus:

```
MCDEF <temp> AS <IF> . . . %WD1. . . . THEN . . .
temp
```

Notes

a. `WD1` was used rather than `D1` since `GE` etc. are macros and it is not desired to call them at this stage.

b. Note that the insert `%WD1.` is not enclosed in literal brackets and is thus inserted when `temp` is defined. Thus if this delimiter were `GR`, then the replacement text of `temp` would be:

```
IF . . . GR . . . THEN . . .
```

and calling `temp` would then accomplish the required call of `IF`.

c. `temp` is enclosed in literal brackets when it is defined in case there is already a `temp` macro in existence. This might arise, for example, if the `WHILE` macro was called recursively.

d. `temp` should be a local macro rather than a global one so that the storage it occupies is released when an exit is made from the `WHILE` macro.

e. This general technique can be used in all cases where it is required to build up a call dynamically. The next Section contains a further example of the technique.

7.4.4 Arithmetic expression macro

Problem

A macro whose name is `(` has been designed so that, when supplied an arithmetic expression as argument, it generates assembly code to calculate the value of the expression and to place the resultant value in an accumulator. This macro will be referred to as the *parenthesis macro*. A typical call of the parenthesis macro might be:

```
(PIG + (Y/6)*Z - 16)
```

This involves a nested call of the same macro. The arguments of the outer call are `PIG`, `(Y/6)`, `Z` and `16`, and the delimiters are `+`, `*` and `-`. It is desired to use this macro to implement a `SET` macro, which allows a macro expression as argument. Calls of `SET` might be:

```
SET DOG = Y
SET VAR = (VAR + 6)/I3 - PIG
```

Solution

The solution to this problem is not to give the `SET` macro a complicated delimiter structure but rather to regard it as a macro with two arguments. The second argument is then passed down to the parenthesis macro, which breaks it down into operators and operands. The `SET` macro is defined:

```
MCDEF SET = NL
AS <MCDEF temp AS <(>%WA2.<)>>
temp
(instruction to store the result in %A1.)
>
```

Notes

a. Notice the use of `temp` to build up a call of the parenthesis macro. In the second of the above examples of `SET`, for instance, `temp` would be defined as:

```
((VAR+6)/13 - PIG)
```

When `temp` was called, it would result in a call of the parenthesis macro with arguments `(VAR+6)`, `13` and `PIG`.

b. It would have been wrong to call the parenthesis macro from within `SET` by writing simply `(%A2.)`, since this would have been interpreted as a call with one argument.

7.4.5 Formal parameter names

Problem

It is desired to use the name `TAXRATE` for the first formal parameter of the macro `DEDUCT`.

Solution

The first part of the definition of `DEDUCT` is written:

```
MCDEF DEDUCT . . . AS <MCDEF TAXRATE AS %A1.
. . .
```

Thereafter within the replacement text of `DEDUCT`, `TAXRATE` can be written in place of `%A1.` whenever the first parameter is required.

7.4.6 Intercepting changes of state

Problem

It is desired in X123 Assembly Language to generate some decimal constants within the replacement text of a macro `SIZE`. However, X123 Assembly Language has two statements, `DECIMAL` and `HEXADECIMAL`, to control the base to which constants are to be written, and this might vary between calls of `SIZE`. Furthermore, it is desired that a call of `SIZE` should not change the base behind the user's back.

Solution

A permanent variable, say `P10`, is used as a switch, the value zero being used to indicate a hexadecimal base. The following is written at the start of the source text:

```
MCSET P10 = 0
MCDEF HEXADECIMAL AS <MCSET P10 = 0
%WD0.>
MCDEF DECIMAL AS <MCSET P10 = 1
%WD0.>
```

and the definition of `SIZE` is written:

```
MCDEF SIZE AS <MCSET T1 = P10
DECIMAL
   .
   .
   .
MCGO L0 IF T1 EN 1
HEXADECIMAL
>
```

thus ensuring that the base is returned to its original state.

Note

This technique is also useful for the following problem: the user has written a macro `SUBS` to generate code for subscripted vectors and it is necessary that `SUBS` generates different code for the two following calls:

a. `LAC SUBS (V, 1)` Load accumulator from element

b. `DAC SUBS (V, 1)` Store accumulator in element

The problem is solved by using the above technique to cause `LAC` and `DAC` to set a switch which the `SUBS` macro can then test to find out which instruction preceded its call.

7.4.7 Remembering code for subsequent insertion

Problem

It is desired to design two macros, `remember` and `insert`, to enable the user to remember text for subsequent insertion. These macros are used in the following way. `remember` is called with a piece of text as argument. `remember` does not generate any code but remembers its argument for subsequent insertion. When the `insert` macro is called, all the pieces of text that have been remembered are inserted.

Solution

A sequence of global macros `I1`, `I2`, ..., `IN` is used, the value of `N` being given by a permanent variable, say `P10`. Each macro represents a piece of text that is to be remembered. The definitions of `remember` and `insert` would be written:

```
MCSET P10 = 0
MCDEF remember ;
AS <MCSET P10 = P10 + 1
MCDEFG I%P10. AS %A1.
>
MCDEF insert AS <MCFOR P1 = 1 TO P10
RECALL I%P1.
REPEAT>
```

where `MCFOR` is the macro of Section 7.4.1 [Macro-time loop], page 76, and `RECALL` is a macro defined thus:

```
MCDEF RECALL NL
AS <MCDEF temp AS %A1.
temp>
```

An alternate solution, which is much faster but which places limitations on the size of each item, can be devised using character variables. This is left as an exercise for the reader, as it is merely a degenerate form of the above solution.

Notes

a. The above solution tries to minimise the amount of storage used. It would have been possible to do without the `RECALL` macro, but this would have involved `N` redefinitions of `temp` within the `MCFOR` loop and so, albeit temporarily, using up rather more storage.

b. Note that the macros `I1` etc. must be global whereas the macro `temp` should be local.

c. An apparently promising technique for this problem which may fail because of excessive use of storage is the following. The entire remembered text is maintained by redefining the `insert` macro as below each time `remember` is called:

```
MCDEF remember ;
AS <MCDEFG <insert> AS insert%A1.
>
```

The trouble with this approach is that old versions of `insert` can never be released, thus using up a very considerable amount of storage.

7.4.8 Constructions with restricted scopes

Problem

It is desired to assign different meanings to a macro `X` within different scopes. One meaning is to apply within the replacement text of a set of macros `M1`, `...` , `MN` whereas another meaning is to apply elsewhere.

Solution

One solution is to redefine `X` as a local macro within each of `M1` to `MN`, but this is tiresome if `N` is large, and slower than the method below even if `N` is one. A better solution is to place the two following definitions at the start of the source text:

```
MCDEFG X . . . AS <{replacement to be used in M1 to MN}>
MCDEF <X . . . > AS <{replacement to be used elsewhere}>
```

The second definition overrides the first. Within the macros `M1` to `MN` the first definition can be re-incarnated by calling `MCNODEF`, which deletes the second definition. Any macros besides `X` that were used within `M1` to `MN` should also be defined as global.

Notes

a. This technique can be used in a variety of applications. It is the best solution in almost all situations where a macro or set of macros has restricted scope, but where this scope does not consist simply of the replacement text of a single macro. Even in the latter case the technique is useful as it is faster than setting up the local definitions every time a macro is called.

b. This technique can be used to extend the technique described in Section 7.4.2 [Examining optional delimiters], page 77, to make it work for name delimiters. For example, if a macro had alternative names `A` and `B`

and, within the replacement text of this macro, it was desired to insert the number `206` if the name was `A` and the number `15` if the name was `B` then this could be achieved, assuming `%` to be an unprotected insert, by writing:

```
MCDEFG A AS 206
MCDEFG B AS 15
MCDEF <OPT A OR B ALL . . .>
AS <. . . MCNODEF%DO. . . .>
```

7.4.9 Optimising macro-generated code

Problem

It is desired to optimise some assembler code generated by ML/I, in particular to cut down possible inefficiencies at the boundary between successive macros.

Solution

There are basically two approaches to producing optimal code:

a. Code can be optimised as it is produced. Typically this would involve using the permanent variables to maintain some sort of indication of the previous instruction(s) generated.

b. A second pass can be made through the macro generated code, to search for various inefficient sequences of instructions.

Except in simple cases, the second method is usually the better. In many machines, considerable optimisation can be performed by maintaining where possible an indication of the contents of the accumulator(s) or other special-purpose registers and thus cutting out redundant loading instructions. This can be done by defining macros to map into numbers all the variables used in the code being generated. A permanent variable, say `P1`, could be used to indicate whether the accumulator was known to contain the current value of a particular execution-time variable. If so, `P1` could contain the number of the variable, otherwise it could be zero. `P1` would need to be zeroised when a label was placed, a subroutine was called, etc. This might be achieved by defining a macro with many alternative names, covering all the situations where the accumulator was clobbered. The macro might be:

```
MCDEF OPT , OR JMS OR ADD OR . . . ALL
AS <MCSET P1 = 0
%WDO.>
```

(assuming that a comma marks the end of a label and `JMS`, `ADD`, etc., are instructions that clobber the accumulator).

7.4.10 Macro to create a macro

Problem

This problem illustrates the use of a macro to set up the definition of another macro. The problem is as follows. It is desired to design a macro `EQUATE` which equates one vector to part of another. Thus the call:

```
EQUATE VEC1 TO VEC2 OFFSET 3
```

would cause each subsequent reference to an element of `VEC1`, which has form, say:

```
VEC1( {subscript} )
```

to be translated into a reference to the corresponding element of `VEC2`, namely:

```
VEC2( {subscript} +3)
```

Solution

The macro `EQUATE` would be defined thus:

```
MCDEF EQUATE TO OFFSET NL
AS <MCDEFG %A1. WITH ( ) AS %A2.(<%A1.>+%A3.)
>
```

Examples

a. The call:

```
EQUATE VEC1 TO VEC2 OFFSET 3
```

would be equivalent to writing the definition:

```
MCDEFG VEC1 WITH ( ) AS <VEC2(%A1.+3)>
```

Note

The main source of error in this sort of problem is to confuse the arguments of the macro that creates the definition with the arguments of the new macro being defined. The rule is that the latter should be enclosed doubly in literal brackets. Hence in the replacement text of `EQUATE`, the arguments within single literal brackets are the arguments of `EQUATE`, which are inserted when the new macro is defined, and the argument within double literal brackets is the argument of the new macro, which is inserted when the new macro is called.

8 Use of system variables

8.1 System variable overview

As previously explained, system variables are introduced by the insert flag `S`. Their purpose is to control certain aspects of the way ML/I operates, and also to report useful information which may be used to change the way that macros operate.

System variables `S1` to `S9` are concerned with the basic operation of ML/I, and are described in this Chapter.

System variables from `S10` upwards are concerned with the way ML/I interacts with its external environment; this is a machine dependent issue, and so the function of these variables may differ considerably between implementations. These system variables are thus described in the relevant Appendix.

8.2 Use of `S1` to `S9`

Some of these variables merely report information, and changing them has no useful effect (except perhaps to confuse the user). Others, as noted, will have a significant effect on the way ML/I operates.

Use of any values other than those explicitly mentioned in this Section will have an undefined effect.

`S1`: Startline control

`S1` controls the insertion of startline characters on input.

- If `S1` has the value zero, no startline characters are inserted; this is the initial setting.
- If `S1` has a value of one, then the special startline character is inserted at the beginning of every subsequent line read from the source text. This character can be processed by ML/I in exactly the same way as any other punctuation character, but is always discarded on output.

`S2`: Controlled line numbers

In many uses of ML/I, some predefined macros are applied to a piece of text. If errors occur, the line numbers in the error messages do not correspond to a listing of the text being processed. For example, if the macros occupy 93 lines, then ML/I takes the first line of the text to be processed as line 94. This can be very confusing to the user of an unfamiliar package of macros.

To remedy this, the source text line number is made accessible to the user by placing it in `S2`.

`S2` has an initial value of zero; ML/I increases it by one at the start of each line of the source text (including the first), and assigns the new value of `S2` to the internal count used in error messages.

The user can change the value of S2 at will, and this will affect the value used in error messages. It will **not** alter the position at which ML/I reads the source text (in other words, it will not perform a seek).

Writers of packages of macros should, at the end of their macros, reset S2 to zero (or whatever value makes the first line of the text to be processed line one—differences can occur when newline is part of a construction name, as ML/I is sometimes looking ahead).

The value of S2 is also useful for other purposes, e.g. for generating unique labels or for use in comments in generated output.

S3: Optional warning markers

If S3 has a value of one, ML/I suppresses the error message normally given if a warning marker is not followed by a macro name.

This is useful if macro calls in the source text are only to be recognised in certain positions, e.g. following a tab or at the start of a line. In such examples the characters 'tab' or 'startline' could be defined as warning markers, and, assuming that not all occurrences need to be followed by macro calls, S3 could be set to one.

Note that, if a warning marker is not followed by a macro name, it is treated as if it were not a construction name at all and is thus normally copied over to the value text. This applies irrespective of whether S3 is being used to suppress the error message.

The following example illustrates how optional warning markers work:

```
MCDEF PIG AS POG
MCINS %.
MCSET S3=1
MCWARN +
+PIG,PIG,MCSET+%S3.+NOTMAC+++
```

would generate the value text:

```
POG,PIG,MCSET+1+NOTMAC+++
```

S4: Option on MCNOTE

If S4 has a value of one, MCNOTE suppresses all the contextual information normally given. All that is output is the value of the argument of MCNOTE, preceded and followed by a newline. For example:

```
MCSET S4 = 1
MCNOTE Message 1
MCNOTE Error in line %S2.
```

would produce the messages:

```
Message 1

Error in line . . .
```

If S4 has a value of zero, the normal contextual information is output.

S5: Count of processing errors

S5 contains the internal count of processing errors encountered so far. This can be useful if the user wishes to check that a particular application has run smoothly, and can be accessed at any time.

S6: Pseudo alphanumeric character

S6 contains a machine dependent character code. This character code specifies a non alphanumeric character which ML/I is to treat as if it were alphanumeric; i.e. not as a separate atom, but potentially part of one. This is useful when processing programming languages which allow other characters within identifiers.

For example, consider the following text:

```
CURRENT_POSITION
```

Normally, ML/I would consider this as three atoms:

a. CURRENT
b. _
c. POSITION

Assuming that the character code for the _ character was 95, then S6 could be set to this value with:

```
MCSET S6=95
```

Subsequently, CURRENT_POSITION would be considered as a single atom.

Initially, S6 contains an implementation dependent value which does not match any character in the character set (typically, this will be −1).

S7: Not used

S7 is not currently used, and its value is undefined.

S8: Not used

S8 is not currently used, and its value is undefined.

S9: Not used

S9 is not currently used, and its value is undefined.

Operation Macro Index

MCALTER 50
MCCVAR 62
MCDEF 45
MCDEFG 48
MCGO 58
MCINS 43
MCINSG 48
MCLENG 53
MCNODEF 47
MCNOINS 47
MCNOSKIP 47
MCNOTE 57
MCNOWARN 47
MCPVAR 61
MCSET 56
MCSKIP 44
MCSKIPG 48
MCSTOP 49
MCSUB 54
MCWARN 42
MCWARNG 48

Concept Index

A

A (as insert flag) 15
ALL keyword 36
ambiguous name 31
argument 6
atom 5

B

B (as insert flag) 15
brackets, literal 18, 27
branch 36

C

call (of macro) 6
call by name 22
capacity 9, 45
case (of characters) 5
causes of error 70
character set 5
character variable 11
clashing names 31
closing delimiter 6
consequences of evaluation 35
construction 20
construction, global 28
construction, local 28
current environment 22, 33
current point of scan 22

D

D (as insert flag) 15
delimiter name 33
delimiter option (on skip) 17
delimiter structure, specifying 33
delimiter, closing 6
delimiter, exclusive 24
delimiter, name 6
delimiter, optional 8
delimiter, repeated 8
delimiter, secondary 6

E

environment 5, 20
environment, current 22, 33
environment, name 20
error messages 63
errors in structure representations 40
errors, causes 70
evaluation 5, 22
evaluation, consequences 35
examples of inserts 15
examples of macros 7
exclusive delimiter 24
expression (macro) 12

F

flag, insert 14
free mode 19
function, system 27

G

global construction 28
global name environment 29

I

initial environment 28, 29, 30
insert 11, 43
insert flag 14
insert name 14
insert, protected 30
insert, unprotected 30
inserted text 15
inserts, examples 15

K

keyword 34, 51
keyword ALL 36
keyword OPT 36
keyword OR 36
keyword WITH 36
keyword WITHS 36

L

label 14, 60
layout character 34
layout keyword 34, 51
list, option 36
literal brackets 18, 27
local construction 28
local name environment 29

M

macro 6, 45
macro call 6
macro element 14
macro expression 12
macro label 14, 60

macro name 6
macro, normal-scan 9
macro, operation 27, 33
macro, straight-scan 9
macro, substitution 27
macro-time statement 10
macro-time variable 10, 11
macros, examples 7
marker, stop 19
marker, warning 19
matched option (on skip) 17, 18
matched skip 18

N

N0 (node zero) 37
name (macro) 6
name clash 31
name delimiter 6
name environment 20
NEC macro 28
nesting 22
node 36
node flag 37
node zero 37
normal-scan macro 9
notation 2

O

operation macro 27, 33
OPT keyword 36
option list 36
optional delimiter 8
OR keyword 36
output text 6
overflow 13

P

permanent variable 11
point of scan, current 22
process 6
protected insert 30
punctuation character 5

R

range 13, 62
recursion 22
repeated delimiter 8
replacement text 6

S

scan, current point of 22
scanned text 5, 22
scanning 22
secondary delimiter 6
skip 17, 44
skip name 17
skip options 17
skip, matched 18
skip, straight 18
source text 6
space character (use of) 71
startline 25
statement, macro-time 10
stop marker 19
straight skip 18
straight-scan macro 9
structure representation 33, 37
structure representations, errors in 40
subroutine 9
subscript 12
substitution macro 27
successor 8
system function 27
system variable 11

T

temporary variable 11
text option (on skip) 17
text, scanned 5
text, value 5

U

unmatched construction 23
unprotected insert 30

V

value text 5
variable, character 11
variable, macro-time 10, 11
variable, permanent 11
variable, system 11
variable, temporary 11

W

WA (as insert flag) 15
warning marker 19
warning mode 19
WB (as insert flag) 15
WD (as insert flag) 15
WITH keyword 36
WITHS keyword 36